Scalable Similarity Queries over Complex Data

Daniel Kocher
00926293

Cumulative dissertation submitted to the Faculty of Natural Sciences of the Paris-Lodron University of Salzburg in partial fulfillment of the requirements for the doctoral degree "Doktor der technischen Wissenschaften (Dr. techn.)".

Supervisor: Univ.-Prof. Dipl.-Ing. Nikolaus Augsten, Ph.D.
Co-Supervisor: Assoc.-Prof. Dr. Ana Sokolova
Department of Computer Sciences
Paris-Lodron University of Salzburg

Salzburg, February 22, 2021

To my family.

ABSTRACT

Our goal is to study and advance the efficient evaluation of similarity queries for complex data. In this thesis, we focus on two types of similarity queries: (1) top-k subtree similarity queries for trees and (2) density-based clustering for sets. A similarity query evaluates predicates that are based on some notion of similarity rather than equality, e.g., two data items are compared using a similarity function such as the overlap similarity for sets or the tree edit distance for trees. The similarity predicates, however, introduce additional complexity during query evaluation and prohibit the usage of standard techniques (like hashing or sorting) to evaluate the respective operators. We develop new index structures and algorithms that are tailored to similarity predicates.

The *top-k subtree similarity query* retrieves the k subtrees in a large document tree that are most similar to a given query tree. The similarity between two trees is assessed using the well-known tree edit distance. Previous solutions are either memory-efficient but slow (i.e., scan the entire document for each query) or fast but require quadratic space in the input size (i.e., an index is built to answer queries fast). We present *SlimCone*, a solution that is based on a linear-space index and allows to retrieve promising subtrees first. Promising subtrees share many node labels with the query tree. SlimCone avoids quadratic space by building relevant parts of the index on the fly. In our experiments on synthetic and real-world data, SlimCone outperforms the state of the art with respect to runtime by up to four orders of magnitude. Furthermore, SlimCone outperforms the index-based solution in terms of memory usage, indexing time, and the number of inspected subtrees.

Density-based clustering is a technique to partition data into clusters, i.e., dense regions that are separated by low-density regions. The popular DBSCAN algorithm recursively expands dense neighborhoods until a low-density neighborhood is reached. It relies on symmetric indexes, i.e., indexes that return all neighbors for a particular point independently of the processing order. For sets, however, the most efficient indexes are asymmetric. An asymmetric index assumes a processing order and returns only a specific part of the neighborhood (all unprocessed neighbors). Thus, they cannot be used in DBSCAN. A baseline that precomputes and materializes all neighbors before executing DBSCAN suffers from a large memory footprint (quadratic in the input size). To the best of our knowledge, we develop the first DBSCAN-compliant solution for sets. Our *Spread* algorithm uses asymmetric indexes and requires only linear space. Spread imposes a processing order on the sets and uses backlinks that keep sufficient information to derive a correct clustering. In our experiments, Spread is competitive with the materialization-based solution in terms of runtime and retains the memory efficiency of DBSCAN. We also present MC-Spread, an extension of Spread to multi-core environments. In our experiments, MC-Spread scales well with the number of cores for datasets with small neighborhoods that are expensive to compute.

ACKNOWLEDGMENTS

First and foremost, I owe my deepest gratitude to my supervisor Nikolaus Augsten who encouraged me to pursue a Ph.D. He introduced me to database research and similarity search, and supported me and my work over the years. His patience, guidance, encouragement, thoroughness, empathy, flexibility, and high research standards were indispensable. Moreover, I would like to thank my co-supervisor Ana Sokolova.

I am also grateful to my colleagues Thomas Hütter (in particular), Mateusz Pawlik, Willi Mann, Bezaye Tesfaye, and Martin Schäler for the numerous worthwhile discussions and the valuable support during my studies. I would also like to acknowledge the contributions of Alexander Miller, Manuel Kocher, Daniel Ulrich Schmitt, Konstantin Thiel, and Manuel Widmoser to the projects in collaboration with Celonis SE. Furthermore, I thank Nikolaus Augsten and Willi Mann for initiating the projects. The great results would not have been possible without you. Likewise, I thank Alfred Egger for his continuous technical support, and Judith Warter for her administrative and personal support throughout the years.

I would like to thank all my friends for their support during my studies although I had little spare time to spend with them. In particular, I would like to thank Christian, Stefan, Heiko, and Evelyn. Your encouragement and support were irreplaceable.

Finally, I would like to express my deepest gratitude to my parents Silvia and Christian, my stepfather Kurt, my siblings Cornelia, Romi, and Manuel, and my grandparents Wilhelmine, Karoline, and Heinz. This thesis would not have been possible without your love, faith, and continuous support.

This work was partially supported by the Austrian Science Fund (FWF): P 29859.

CONTENTS

Abstract iii
Acknowledgments v
List of Figures ix
List of Tables xi
List of Algorithms xii
1 INTRODUCTION 1
1.1 Objective of this Thesis . . . 1
1.2 Similarity Queries . . . 2
1.3 Data Representation & Similarity Functions . . . 6
1.3.1 Trees & the Tree Edit Distance . . . 6
1.3.2 Sets & Set Similarity . . . 8
1.3.3 Practical Use Case . . . 12
1.4 Contributions . . . 13
1.4.1 Scalable Top-k Subtree Similarity Queries . . . 13
1.4.2 Density-Based Clustering for Sets . . . 14
1.4.3 Reproducibility . . . 14
1.5 Thesis Outline . . . 15
2 A SCALABLE INDEX FOR TOP-K SUBTREE SIMILARITY QUERIES 17
2.1 Introduction . . . 17
2.2 Notation, Background, and Problem Statement . . . 19
2.3 Effective Candidate Generation . . . 21
2.4 Index and MergeAll Algorithm . . . 23
2.4.1 Candidate Index . . . 23
2.4.2 MergeAll Algorithm . . . 25
2.5 Cone: Partition-Based Traversal . . . 29
2.6 Linear Space Index and SlimCone . . . 32
2.6.1 Indexing in Linear Space . . . 32
2.6.2 The SlimCone Algorithm . . . 33
2.7 Efficient Index Updates . . . 37
2.8 Related Work . . . 41
2.9 Empirical Evaluation . . . 42
2.9.1 Setup & Datasets . . . 42
2.9.2 Indexing . . . 44
2.9.3 Effectiveness and Query Time . . . 45
2.10 Conclusion . . . 48
3 SCALING DENSITY-BASED CLUSTERING TO LARGE COLLECTIONS OF SETS 51
3.1 Introduction . . . 51
3.2 Background & Problem Definition . . . 54
3.2.1 Set Similarity and ϵ-Neighborhood . . . 54
3.2.2 Indexing Techniques for Sets . . . 55

3.2.3 Density-Based Clustering . . . 56
3.2.4 The DBSCAN Algorithm . . . 57
3.2.5 Problem Statement . . . 57
3.3 Baseline Approaches . . . 58
3.3.1 Sym-Clust: DBSCAN with Inverted Index . . . 58
3.3.2 Join-Clust: Materialized Neighborhoods . . . 60
3.4 The Spread Algorithm . . . 62
3.4.1 Key Challenges . . . 62
3.4.2 Data Structures . . . 64
3.4.3 The Algorithm . . . 64
3.4.4 Correctness . . . 65
3.4.5 Complexity Analysis . . . 67
3.5 Experimental Evaluation . . . 68
3.5.1 Index & Cluster Statistics . . . 69
3.5.2 Runtime Efficiency . . . 69
3.5.3 Memory Efficiency . . . 70
3.5.4 Scalability . . . 73
3.6 Related Work . . . 74
3.7 Conclusion . . . 75
4 A MULTI-CORE SOLUTION FOR DENSITY-BASED CLUSTERING OF SETS 77
4.1 Preliminaries . . . 78
4.2 A Simple Multi-Core Extension of Spread . . . 80
4.3 Refining the Simple Algorithm . . . 81
4.3.1 Idle Clustering Thread . . . 81
4.3.2 Cache Misses and False Sharing . . . 82
4.3.3 Controlling the Memory . . . 83
4.4 Multi-Core Spread . . . 84
4.5 Multi-Core Join-Clust . . . 87
4.6 Experimental Results . . . 88
4.7 Conclusion & Outlook . . . 98
5 CONCLUSIONS & FUTURE WORK 101
A REPRODUCIBILITY PACKAGE 105
A.1 Hardware, Operating System, and Software . . . 105
A.2 Quick Start . . . 105
A.3 Reproducibility Package . . . 106
A.3.1 Datasets, Queries, and Results . . . 106
A.3.2 Package Details . . . 106
A.4 Flexibility . . . 108
A.4.1 Parameters . . . 108
A.4.2 Plots . . . 109
A.5 Time Estimates . . . 109
Bibliography 110

LIST OF FIGURES

Figure 1.1 Three different data items that represent the same address "500 S Buena Vista St, Burbank, CA 91521". 3
Figure 1.2 Types of similarity queries studied in this thesis. 4
Figure 1.3 Signature-based inverted list index for eight data items. Two signatures are generated per data item, resulting in five distinct signatures over all data items. 5
Figure 1.4 Ordered vs. unordered labeled trees. 7
Figure 1.5 The node edit operations between ordered labeled trees. . . . 7
Figure 1.6 Example text as binary vector and set. 9
Figure 1.7 Prefix filtering exemplified for two sets, r and s, and a prefix length of 3. 11
Figure 1.8 Example process and its representations. 12
Figure 2.1 Running example. 22
Figure 2.2 Baseline index structure for document T of our running example (cf. Figure 2.1). 24
Figure 2.3 Worst-case document for the inverted list index (root to the left, leaf to the right). 24
Figure 2.4 Stripes and partitions w.r.t. query Q. 25
Figure 2.5 MergeAll after processing stripes $j = 2$. 27
Figure 2.6 Cone traversal of the inverted list index in candidate score order. 30
Figure 2.7 Processed subtrees of Cone. 32
Figure 2.8 Linear-space index for example document T. 34
Figure 2.9 Finding the starting point of a slim list. 35
Figure 2.10 Slim lists, path caches, and path ends. 36
Figure 2.11 Index update example. 40
Figure 2.12 Build time, index size, and update time. 45
Figure 2.13 Merge, Cone, Slim: Query time and number of verifications over document size, $k = 10$, $|Q| = 16$. 46
Figure 2.14 State of the art vs. Slim: Query time and number of verifications over document size, $k = 10$, $|Q| = 16$. 46
Figure 2.15 State of the art vs. Slim: Query time and number of verifications over query size $|Q|$, $k = 10$. 47
Figure 2.16 State of the art vs. Slim: Query time and number of verifications over varying result size k, $|Q| = 16$. 48
Figure 3.1 Symmetric candidates with ϵ-neighbors (blue); asymmetric candidates with lookahead neighbors (red). 53
Figure 3.2 Symmetric and asymmetric prefix index, $\epsilon = 3$. 55
Figure 3.3 Running example, $\epsilon = 3$, minPts = 4. 58
Figure 3.4 Redundant neighborhood queries. 59

Figure 3.5 Symmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi = 4$. 60
Figure 3.6 Asymmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi^i = 2$. 60
Figure 3.7 Runtime over ϵ, minPts = 16. 71
Figure 3.8 Runtime over minPts, $\epsilon = 3$. 71
Figure 3.9 Runtime over data size, $\epsilon = 3$, minPts = 16. 71
Figure 3.10 Main memory over ϵ, minPts = 16. 72
Figure 3.11 Main memory over minPts, $\epsilon = 3$. 72
Figure 3.12 Main memory over data size, $\epsilon = 3$, minPts = 16. 73
Figure 3.13 Backlinks peak over ϵ, minPts = 16. 73
Figure 3.14 Backlinks peak over minPts, $\epsilon = 3$. 73
Figure 4.1 Wallclock time for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16. 82
Figure 4.2 CPU cycles for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16. 82
Figure 4.3 Cache misses for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16. 83
Figure 4.4 Wallclock time over the number of cores, $\epsilon = 2$, minPts = 16. . 90
Figure 4.5 Wallclock time over the number of cores, $\epsilon = 3$, minPts = 16. . 91
Figure 4.6 Wallclock time over the number of cores, $\epsilon = 4$, minPts = 16. . 91
Figure 4.7 Wallclock time over the number of cores, $\epsilon = 5$, minPts = 16. . 92
Figure 4.8 Main memory over the number of cores, $\epsilon = 2$, minPts = 16. . 94
Figure 4.9 Main memory over the number of cores, $\epsilon = 3$, minPts = 16. . 94
Figure 4.10 Main memory over the number of cores, $\epsilon = 4$, minPts = 16. . 95
Figure 4.11 Main memory over the number of cores, $\epsilon = 5$, minPts = 16. . 95
Figure 4.12 CPU cycles for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16. 96
Figure 4.13 Cache misses for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16. 96
Figure 4.14 Main memory over memory constraint, 8 cores, $\epsilon = 5$, minPts = 16. 97
Figure 4.15 Wallclock time over memory constraint, 8 cores, $\epsilon = 5$, minPts = 16. 97

LIST OF TABLES

Table 1.1 Similarity resp. distance functions for two sets, r and s, and a similarity resp. distance threshold t. 11
Table 2.1 Notation overview. 21
Table 2.2 Example subtrees ordered by candidate score. 23
Table 2.3 Dataset characteristics. 43
Table 3.1 Notation overview. 57
Table 3.2 Characteristics of datasets. 68
Table 3.3 Index & cluster statistics for $\epsilon = 3$, minPts = 16. 70
Table 4.1 Speedups (and the corresponding number of cores) for each dataset and $\epsilon \in \{2, 5\}$ (NA . . . not available); speedup of algorithm A is $speedup(A) = \frac{\text{1-core time of } A}{\text{max-core time of } A}$. 93

LIST OF ALGORITHMS

1 MergeAll(Q, T, k) . . . 28
2 Process-Subtree($T_i, lb, \mathcal{B}$) . . . 28
3 Verify-Bucket($\mathcal{B}$) . . . 28
4 Cone(Q, T, k) . . . 33
5 Process-List($l_\lambda, \mathcal{B}$) . . . 33
6 SlimCone(Q, T, k) . . . 38
7 Process-Smaller($l_\lambda, \mathcal{B}$) . . . 39
8 Process-Larger($l_\lambda, \mathcal{B}$) . . . 39
9 Materialize-Neighborhoods(R, ϵ) . . . 61
10 Create-Index (R, ϵ) . . . 61
11 Probe (r, I, ϵ) . . . 61
12 Verify-Pair (r, s, ϵ, po) . . . 62
13 Spread(R, ϵ, minPts) . . . 66
14 MC-Spread(R, ϵ, minPts) . . . 84
15 Compute-Neighborhoods($R, \epsilon, I, ln, next_id$) . . . 85
16 Probe-Notify(R, ϵ, I, ln, id) . . . 85
17 MC-Cluster($R, \epsilon, minPts, I, ln, next_id$) . . . 85
18 Cluster(r, minPts, ln, ds, nc_bl, c_bl) . . . 86
19 MC-Materialize-Neighborhoods(R, ϵ) . . . 88
20 Compute-Pairs(R', ϵ, I) . . . 88

1 INTRODUCTION

Database systems must cope with an ever-growing amount of data and serve applications that require the storage and retrieval of complex data items. For example, hierarchical or tree-structured data are used in bioinformatics [3, 5, 57], natural language processing [77], and pattern recognition [68]; set-valued attributes are used to represent objects from different domains including tweets [82, 108], user groups [84], or tags [23]. Database systems support a wide range of operators, and queries using these operators must scale to large data corpora. One important class of queries are *similarity queries*. Similarity queries evaluate similarity predicates that compare data items using some notion of similarity instead of checking for equality [11]. The similarity of two data items is assessed using a similarity function. Various similarity functions for different data types have been proposed, e.g., the edit distance for strings [75] and trees [109], the Jaccard similarity for sets [61], or the Hamming distance for binary vectors [52]. Well-known operators like the join operator can be extended to support similarity predicates.

The similarity predicate introduces additional complexity into the evaluation of the respective operator, and new challenges arise. For example, hash joins and sort-merge joins, two well-known approaches to evaluate equality joins, cannot be readily applied to compute a similarity join [10]. Therefore, specialized index structures and algorithms must be developed to support efficient and effective similarity queries in database systems.

The remainder of this chapter is organized as follows. Section 1.1 presents the objective of this thesis. In Section 1.2, we introduce similarity queries, the two query types that are the focus of this thesis, and discuss a commonly used signature-based filter-verification framework. We discuss selected representations of complex data items (i.e., ordered labeled trees and sets), similarity functions, and a practical use case from industry (among other application areas) in Section 1.3. Section 1.4 summarizes the contributions of this thesis and points to published work of the author that was not included into this thesis. Finally, Section 1.5 outlines the remainder of this thesis.

1.1 OBJECTIVE OF THIS THESIS

In this thesis, we study indexes and algorithms for similarity queries over complex data. In particular, we focus on two popular query types: (1) The *top-k subtree similarity query* for ordered labeled trees and (2) the *density-based clustering of sets*. The goal of this thesis is to develop solutions that scale to large datasets with respect to runtime and main memory consumption.

There are many representations to store complex data items in a database system, for example, ordered labeled trees, binary vectors, or sets. So-called *similarity* or *distance*

functions can be used to assess the similarity between two data items of a particular representation. The *tree edit distance* [109] is a widely used measure to assess the similarity between two ordered labeled trees. Allowing three node edit operations, the tree edit distance is defined as the minimum number of edit operations that transform one tree into the other. In the case of binary vectors, the Hamming distance [52] is a commonly used distance measure [78, 87, 110]. For sets, many different similarity functions such as the overlap similarity [130] or the Jaccard similarity [61] have been proposed.

Similarity functions can be used in combination with various database operators such as selections, sorting, or joins. A naive solution replaces the equality operator with an operator that computes the similarity. In many scenarios, however, this results in inefficient queries because (i) distance/similarity functions are often computationally expensive and/or (ii) representation-specific optimizations are not leveraged. (i) Consider, for example, the current state-of-the-art solution AP-TED$^+$ by Pawlik and Augsten [91] for the exact tree edit distance. It runs in $O\left(n^3\right)$ time and $O\left(n^2\right)$ space with n being the number of nodes in the tree, and has shown to be worst-case optimal. A nested-loop join with the tree edit distance in the join predicate, however, is infeasible for large collections of trees because the cubic distance algorithm must be executed for each pair of trees [59]. (ii) Specialized index structures and processing techniques often provide much faster runtimes, for example, in a set similarity join scenario [6, 16, 29, 36, 37, 45, 80, 81, 108, 122, 124, 130]. Furthermore, the scalability of a query may be limited with respect to other dimensions, for example, a large memory footprint or limited parallelism. The goal is to scale queries in all dimensions.

1.2 SIMILARITY QUERIES

Database systems store and organize large amounts of data such that users can query the data. A database system aims to answer a given query efficiently. From a user's perspective, a query should be answered almost in real time. Modern database systems have to deal with large amounts of data. To this end, storing the data is not the only problem but also providing good query performance for online transaction processing (OLTP) and online analytical processing (OLAP) workloads [101]. Most database systems are able to provide good performance for *exact* queries, i.e., queries that use an equality predicate to compare two items in the database. A typical example for an exact query is finding a person by a unique number like the social security number. Sometimes, however, it may be impossible to formulate an exact query that provides the user with a satisfying answer. Consider, for example, a user that wants to find a particular address in an address database. First of all, the same address may be stored in different variations, e.g., abbreviations can be used ("S Buena Vista Street" vs. "S Buena Vista St"), dashes could be replaced by whitespaces ("Jakob-Haringer-Strasse" vs. "Jakob Haringer Strasse"), or there may be typos ("S Buena Vista St" vs. "S Buema Fista St"). Secondly, different data items may refer to the same address due to their internal representation [11]. Figure 1.1 shows three reasonable tree representations of the same address "500 S Buena Vista St, Burbank, CA 91521". Representation 1 contains

one node per address part (split by ","), whereas Representation 3 additionally splits the numerical and textual components. Contrarily, Representation 2 simply stores the entire address in a single node. An exact query finds at most one of them, and none if the representation of the query tree is not identical to one of these three representations. Similarity queries aim to solve exactly this problem.

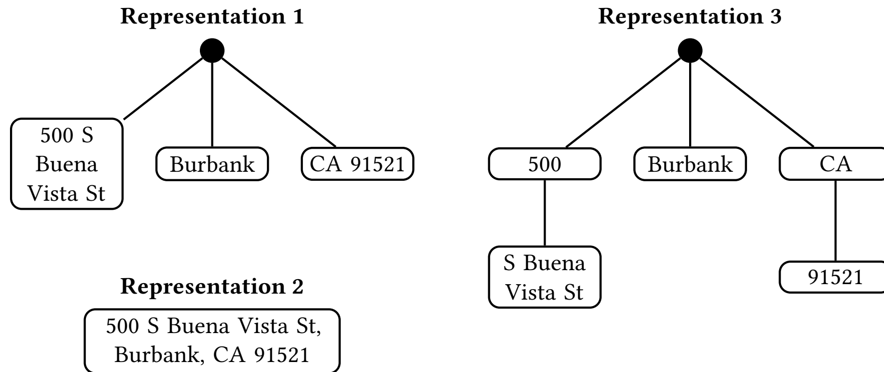


Figure 1.1: Three different data items that represent the same address "500 S Buena Vista St, Burbank, CA 91521".

In contrast to an exact query, a similarity query uses a similarity predicate to match data items. Typical similarity predicates include similarity functions or distance functions. The complexity of a similarity function differs depending on the representation of the data, e.g., comparison of trees with the exact tree edit distance requires cubic time [91], whereas the overlap between (sorted) sets can be computed in linear time [81]. Relevant data representations and popular similarity functions are discussed in Section 1.3.

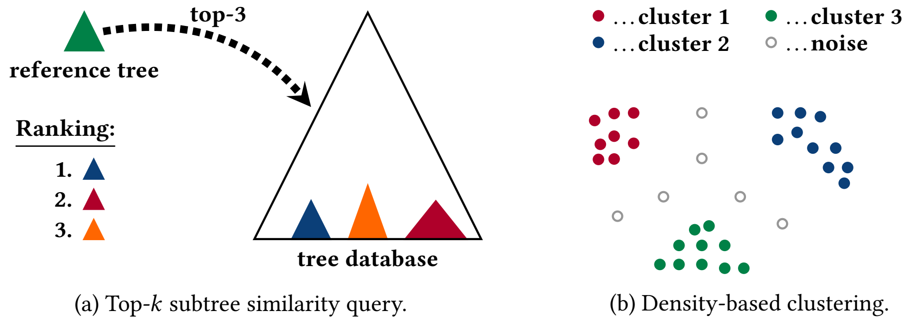
TYPES OF SIMILARITY QUERIES Among the most popular types of similarity queries are *range*, *nearest neighbor*, and *ranking* queries. Given an attribute A and two boundaries L and U (lower and upper), a *range* query retrieves all data items for which the attribute is within the lower and upper end, i.e., $L \leq A \leq U$ holds [101]. For example, a user can ask for all addresses with a house number between 500 and 600. One specific type is the *ϵ-range* (or region) query, which finds all data items (i.e., an arbitrary number of data items) that satisfy a given similarity threshold, ϵ, with respect to a given reference item and a particular similarity function [101]. Neighboring data items are said to be *neighbors* or *in the neighborhood*. Instead, *nearest neighbor* queries aim to find the data item that is most similar to a given reference item and a specific similarity function (i.e., its nearest neighbor) [101]. For example, a user can ask for the street name in Los Angeles that is most similar to "S Buena Vista St". One generalization of the nearest neighbor query is the *k-nearest neighbor* (k-NN) query, which finds the k most similar neighbors (instead of only a single, most similar neighbor) [39]. Finally, a *ranking* query retrieves data items in a particular order that reflects the similarity of the data items with respect to a given reference item. If the number of ranked data items is further restricted by a user-defined numeric value, k, this is called *top-k* query [60]. It ranks up to k data items that are most similar to a given reference item (typically

in ascending similarity to the reference item). For example, a user can ask a database system to rank the 10 street names that are most similar to "S Buena Vista St".

This thesis covers two specific types of similarity queries, namely (1) top-k subtree similarity queries for ordered labeled trees and (2) density-based clustering for sets.

(1) A top-k subtree similarity query finds and ranks the k most similar subtrees in a large tree database with respect to a given reference tree [8, 31]. Figure 1.2a shows the principle of a top-k subtree similarity query for $k = 3$: The query returns three trees (blue, orange, red) of the tree database that are most similar to the reference tree (green). Tree representation and the comparison of trees are covered in Section 1.3.1.

(2) Density-based clustering partitions a collection of data items into dense regions, so-called *clusters*, that are separated by regions of low density [40, 100]. The user has to specify two input parameters, minPts and ϵ: (i) minPts is the minimum number of data items in a neighborhood to consider it be dense, and (ii) ϵ is the radius of the neighborhood, i.e., the similarity threshold for the respective ϵ-range query. Typically, ϵ-range queries are used to identify data items that satisfy the density requirement, i.e., data items that have at least minPts neighbors within a radius of ϵ are in a dense region. These data items are then recursively expanded until a data item with less than minPts neighbors is encountered [40]. Figure 1.2b shows the intuition of the density-based clustering operation: Given a collection of data items in 2D space (for simplicity), there are three dense regions (red, green, blue) and a region of low density (gray). Each dense region forms a cluster, and data items in the low-density region are *noise*, i.e., they are considered irrelevant.



(a) Top-k subtree similarity query.

(b) Density-based clustering.

Figure 1.2: Types of similarity queries studied in this thesis.

INDEX STRUCTURES & FRAMEWORK Index structures (or indexes) are a key component of database systems to offer good performance. On a high level, an index is a data structure that provides shortcuts to data items a query has to find. For many algorithms the index is a black box that takes a request and returns all data items that satisfy the request. An example request is to find all data items that represent a specific street in Los Angeles. In the context of this thesis, we focus on *signature-based inverted list indexes*, which are at the core of our solutions.

A signature-based inverted list index (inverted index, for short) computes a signature for every data item. A signature function maps a data item to one or multiple (numerical) hash values. A typical approach in similarity search is (i) to compute multiple

signatures for each data item and (ii) to design the signature function such that two similar data items share at least one signature [16, 130]. The inverted index maintains one list per signature. The inverted list of a particular signature sig_i stores all data items that produce sig_i. This organization simplifies finding all data items for a particular signature sig_i as it only requires a single lookup (typically done in constant time). In Section 1.3.2, we briefly discuss common signatures for set similarity algorithms.

The indexes are used in a *filter-verification* framework [81], which is a common algorithmic design pattern in similarity search. After building the index, we perform the following steps to find items that are similar to a given data item d: (i) All signatures for d are computed, denoted $Sig(d) = \{sig_1, sig_2, \ldots, sig_m\}$, $|Sig(d)| = m$. (ii) An index lookup is performed for every signature $sig_i \in Sig(d)$, $1 \leq i \leq m$. We say d is *probed* against the index. The data items of all lists, l_{sig_1} to l_{sig_m}, are collected (cf. Figure 1.3) and form the *candidates* of d. The candidates may contain false positives, i.e., data items that share a signature with d but are not similar to d. Notably, the candidates do not contain false negatives, i.e., all data items similar to d appear in at least one list. (iii) Each candidate is then verified against d. The verification procedure evaluates the similarity predicate of the query (typically by computing a similarity function) and discards the false positives. Candidates that pass the verification procedure form the final result.

Figure 1.3 shows an example dataset with eight data items, each of which generates two signatures (resulting in a total number of five distinct signatures over all data items). Consider, for example, data item d_2 in Figure 1.3: d_2 generates the signatures sig_2 and sig_3 (this can be inferred from the fact that d_2 appears in these lists). After a lookup of these two lists, we get the candidates $\{d_2, d_3, d_4, d_7, d_8\} \cup \{d_1, d_2, d_3\} = \{d_1, d_2, d_3, d_4, d_7, d_8\}$, which must be verified in order to eliminate false positives. Note that d_5 and d_6 are not part of the candidates since they generate neither sig_2 nor sig_3.

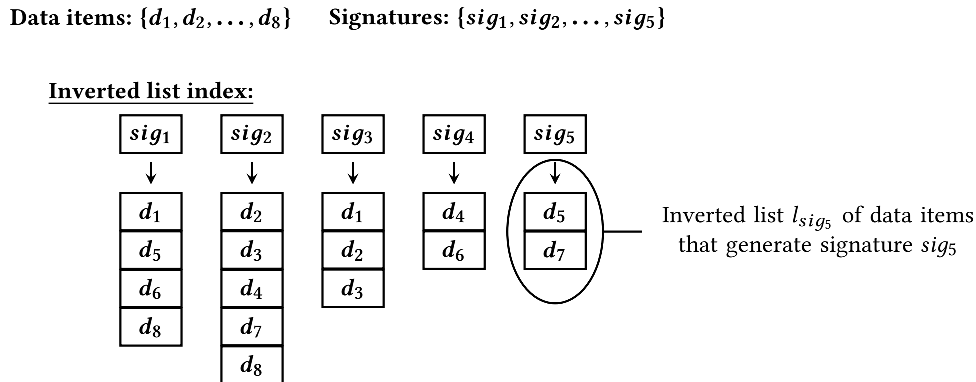


Figure 1.3: Signature-based inverted list index for eight data items. Two signatures are generated per data item, resulting in five distinct signatures over all data items.

1.3 DATA REPRESENTATION & SIMILARITY FUNCTIONS

Database systems are used in a wide range of applications, operating on different types of data. Internally, complex data items are often mapped to some abstract representation, e.g., hierarchical data are represented as trees [5, 19, 106, 136]. The similarity function used in the similarity predicate is typically specific to the representation of the underlying data. Furthermore, specific optimization and indexing techniques have been developed for different representations. In the course of this section, we cover selected data representations as well as the corresponding similarity and distance functions. We focus on exact techniques and do not discuss approximations, which have been proposed for some types of similarity queries [13, 49, 128]. Finally, we discuss a practical use case from industry.

1.3.1 *Trees & the Tree Edit Distance*

In this section, we discuss a typical representation of hierarchical data, the *ordered labeled tree*, and a widely used similarity measure for trees, the so-called *tree edit distance*.

ORDERED LABELED TREES Hierarchical data encodes hierarchical dependencies between (parts of) data items. Application areas that use hierarchical data include, among others, biology and bioinformatics [3, 5, 57], pattern recognition and image analysis [17, 68], automatic information extraction [67, 97], and similarity queries [47, 59, 125]. Various data formats have been developed to represent hierarchical data, e.g., XML [88], JSON [24], or the Asterix Data Model (ADM) [4]. Trees can be used to represent hierarchical data in a database system. A tree T is a rooted, directed, acyclic, connected graph with nodes $V(T)$ and directed edges $E(T) \subseteq V(T) \times V(T)$, where the *root* node has no inbound edge and *leaf* nodes have no outgoing edges. Due to the hierarchical nature of trees, an edge $(u, v) \in E(T)$ represents a parent-child relationship between the two nodes u and v, that is, u is the *parent* of v, and v is the *child* of u. Nodes $v_1, v_2, \ldots, v_k$ that share a common parent are called *siblings*. A tree also consists of transitive parent-child relationships: If a path from the root node to some node v contains another node u, $u \neq v$, u is an *ancestor* of v, and v is a *descendant* of u. A *subtree* T_u of T is a tree that is rooted at node u and consists of all descendants of u as well as all edges that connect this subset of nodes in T.

A tree is called *labeled* if a label is associated with each node $v \in V(T)$. Labels carry the data of a node, e.g., a string or a numeric value. In this thesis we assume node-labeled trees.

We distinguish ordered and unordered trees. In an *unordered tree*, the order of child nodes does not matter, whereas *ordered* trees define a strict, total order on the children of a node [11]. As will be covered later in this section, the existence of an order heavily affects the computational complexity of the tree edit distance. Figure 1.4 illustrates two trees that represent the same point of interest (POI), but with different child orderings for the "Address" node. If we interpret the trees as unordered, trees T_0 and T_0' are

identical. If interpreted as ordered trees, however, T_0 and T_0' are not identical since they differ in the sibling order.

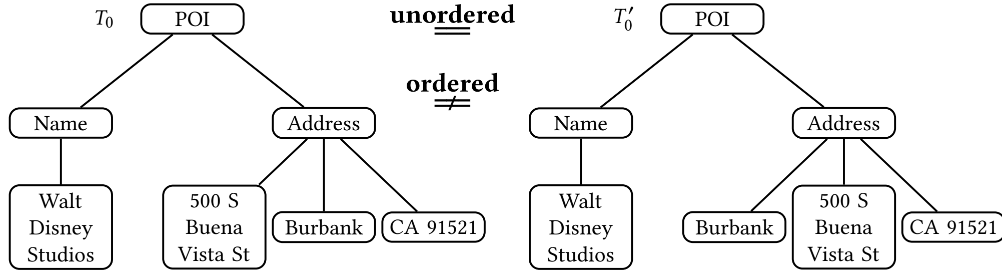


Figure 1.4: Ordered vs. unordered labeled trees.

TREE EDIT DISTANCE The *tree edit distance* assesses the similarity of two ordered (or unordered) labeled trees. For ordered labeled trees, the tree edit distance is based on three *node edit* operations [136]: (1) *Rename* updates the label of a given node u. (2) *Delete* removes a given node u and the children of u become children of its parent (starting at u's sibling position and retaining their order). (3) *Insert* embeds a new node u between an existing node p and a consecutive (possibly empty) subsequence of p's children.

Figure 1.5 illustrates the node edit operations. Starting from the leftmost tree, we follow the edit operations in order (1) to (4). (1) Renaming "500" to "700" changes the label of a single node in T_1, resulting in tree T_2. (2) The deletion of "CA" from T_2 changes the structure of the tree, i.e., the resulting tree T_3 has one node less. (3) Inserting a node with label "CA" in between "Address" and "91521" reverts this structural change, and (4) renaming "700" to "500" results in the original tree, T_1. We observe that insertion and deletion are inverse (structural) operations.

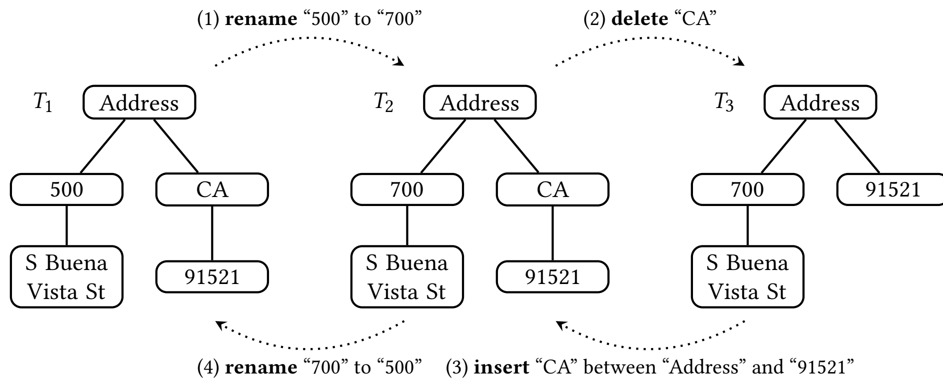


Figure 1.5: The node edit operations between ordered labeled trees.

Most tree edit distance algorithms decompose the input trees into smaller subtrees and subforests based on a recursive formula. For ordered labeled trees, Tai [109] proposed the first algorithm to compute the tree edit distance in $O(n^6)$ time, where n denotes the number of nodes in the larger of the two trees. Since then, the tree

edit distance was subject to many improvements. A popular algorithm was proposed by Zhang and Shasha [136] in 1989, which improved the time and space complexity to $O(n^4)$ and $O(n^2)$, respectively. To date, the best known algorithm is AP-TED$^+$ proposed by Pawlik and Augsten [91]. AP-TED$^+$ is worst-case optimal and runs in $O(n^3)$ time and $O(n^2)$ space. We refer to Pawlik and Augsten [91] for a detailed description of AP-TED$^+$. The tree edit distance for unordered labeled trees has been shown to be NP-complete [137] and is out of the scope of this thesis.

Chapter 2 of this thesis presents a scalable solution to answer top-k subtree similarity queries for ordered labeled trees under the tree edit distance. The tree edit distance computation is used as a black box in this work, i.e., any algorithm can be used.

1.3.2 *Sets & Set Similarity*

Sets are a common representation of complex objects for the sake of similarity computation, e.g., strings are modeled as sets of q-grams (substrings of length q) [117], trees are represented as sets of pq-grams (subtrees of a particular form) [12], or text documents are interpreted as sets of words [123]. This section covers sets and their representation as binary vectors, revisits similarity measures for sets, and discusses popular signatures that are used in algorithms for set similarity.

SETS & BINARY VECTORS In general, a *set* is a group of unique elements in arbitrary order (if elements are not unique, it is called *multiset*). In our context, a set often represents a single data item, and an element of a set is commonly referred to as *token*. For a collection of sets, R, the union of distinct tokens over all sets is called the token *universe*, U, and the number of tokens in the token universe is the *universe size*, $|U|$. Data from many diverse domains can be represented as sets [81], for example, purchases in online shops (one token per product category) [138], image meta-data (one token per tag) [23], online user behavior[1] (one token per clicked link), user meta-data (one token per user interest or group membership; one token per listened track or watched movie) [84, 94], or business event logs (one token per transition between two activities) [70]. In practice, the tokens of a set are often sorted with respect to some total order that is consistent over all sets, e.g., with respect to the global token frequency (GTF). Sorting the tokens simplifies the comparison of sets and enables other optimization techniques for set similarity algorithms [6, 29, 81, 130]. Typically, a set is stored as a sequence of unique integer tokens, where an integer represents an element in the set.

Binary vectors (also called *bit vectors* or *bitmaps*) are a common representation method for sets [96, 118]. More formally, a binary vector can be interpreted as a hash function that maps an arbitrary domain to the binary domain $\{0, 1\}$. Effectively, a set is then represented as a d-dimensional binary vector with $d = |U|$. A bit at position i is (un)set if dimension i is (not) present in the data item. Although d might be large, one rationale to represent sets as binary vectors is space because each dimension can be expressed by a single bit. Furthermore, effective and efficient compression techniques

1 http://fimi.uantwerpen.be/data/

for binary vectors have been developed (e.g., Roaring bitmaps [27, 28, 74] or tree-encoded bitmaps [73]) and modern hardware provides very efficient (advanced) bitwise operations, which can be used to speed up the computation of similarity functions for binary vectors (e.g., the `POPCOUNT` operation [103] to count the number of distinct positions in two binary vectors). Semantically, binary vectors and sets can be used to represent the same data item. Which representation to use in practice depends on the population of a binary vector. For a large universe size and sparse binary vectors, the set representation may require less memory compared to binary vectors. In a nutshell, this is also the mechanism behind the Roaring bitmap compression technique [27, 28, 74]: Depending on the population characteristics of a binary vector, the data item is stored differently, i.e., the binary vector is split into chunks of 2^{16} bits and each chunk is stored in the most memory-efficient way (chosen from three possible representations, including sets and plain binary vectors).

Figure 1.6 shows a common use case for binary vectors and sets: similarity queries in texts. First, a tokenization is applied to the text, i.e., the text is split based on a particular splitting policy. In this example, the text is split by whitespace and the tokenization provides us with 17 string tokens (words). The universe size is the number of *distinct* string tokens in the entire text, which is 15 in this example ("one" and "only" each appear twice). Second, the string tokens in the universe are mapped one-to-one to unique integers, e.g., "Lord" is mapped to 2 (denoted Lord $\longrightarrow$ 2). The string-to-integer mapping is stored in a so-called *translation table*. Consider a binary vector for the highlighted portion of the text "Lord of the Ring". Each string token is mapped to an integer and the corresponding bit in the binary vector of size 15 (universe size) is set. The equivalent set representation $\{2, 4, 8, 9\}$ is also shown in Figure 1.6. We typically store tokens in their integer representation because integers are computationally cheaper to compare than strings.

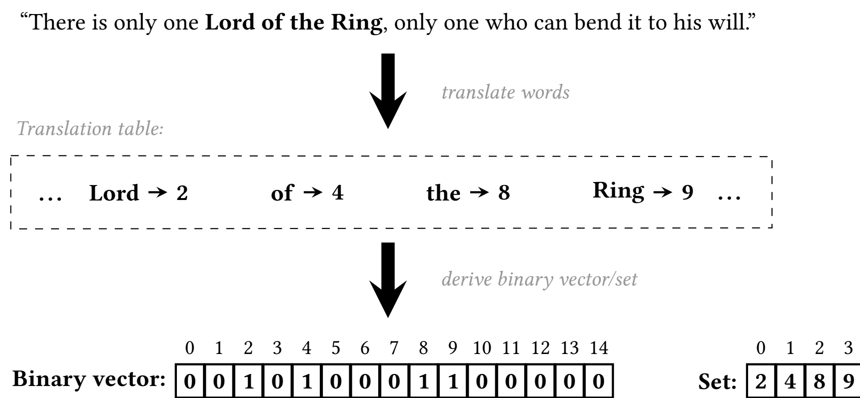


Figure 1.6: Example text as binary vector and set.

SET SIMILARITY & DISTANCE FUNCTIONS Most similarity measures for sets are based on the intersection between the tokens of the sets [11]. Two sets, r and s, are considered similar if they share many tokens, i.e., the intersection $|r \cap s|$ is large. Similarity functions are often normalized to the interval $[0, 1]$, and similar sets

have a similarity value closer to 1 than non-similar sets. In the case of (normalized) distance functions, which are also based on the set intersection $|r \cap s|$, identical sets have a distance of 0 (and the distance of dissimilar sets is close to 1). Typically, both normalized and non-normalized similarity resp. distance functions are applicable in the context of similarity queries. Table 1.1 summarizes popular set similarity and set distance functions. The overlap similarity [130], $O(r, s)$, computes the size of the intersection $|r \cap s|$ and ranges from 0 (no shared token) to $\min\{|r|, |s|\}$ (identical sets or subset relationship). Jaccard [61], Cosine [130], and Dice [38] similarity are different normalizations of the overlap similarity to the interval $[0, 1]$, where 0 denotes dissimilarity and 1 denotes identity. The Hamming distance [52] is a well-known distance measure for binary vectors (of equal size) and denotes the number of bit positions in which the vectors differ. In the context of sets, the Hamming distance denotes the number of tokens that exist in only one of the sets, and ranges from 0 (identical sets) to $|r| + |s|$ (no shared token) [11]. Table 1.1 provides the definitions of the various similarity and distance functions as well as the so-called equivalent overlap, *Eq. Overlap*, and a size lower bound [6, 81], *Min. Size*, which are discussed below. Moreover, the rightmost column, *Norm.*, indicates whether the respective function is normalized.

Many similarity search algorithms for sets are based on a similarity threshold. Set similarity join algorithms, for example, aim to find all similar pairs in two collections of sets, R and S, that exceed a given threshold t with respect to a user-defined similarity function $sim(r, s)$, i.e., the join result is $\{(r, s) \mid r \in R, s \in S, sim(r, s) \geq t\}$ [81]. Note that (i) the similarity function can be any of the (normalized) similarity functions listed in Table 1.1, but also the overlap similarity, $O(r, s)$, and (ii) we can also use a distance function (e.g., the Hamming distance) in combination with a minimum distance as threshold, then the join result is $\{(r, s) \mid r \in R, s \in S, H(r, s) \leq t\}$ for a given Hamming distance threshold t.

Threshold-based algorithms for set similarity often translate the given similarity threshold into an *equivalent overlap* that suffices to decide whether two sets r and s are similar. Computing the overlap between r and s is typically much cheaper than computing the respective similarity function. Furthermore, the overlap computation (during verification) can stop once we know that $|r \cap s|$ is larger than or equal to the required equivalent overlap [81]. This is due to the fact that we are not interested in the true similarity of r and s, but we only want to know whether r and s satisfy the given threshold. Likewise, the *size lower bound* (min. size) can be exploited by set similarity algorithms: If the size difference of r and s is too large, then r and s cannot satisfy the given similarity threshold [6]. Both equivalent overlap and size lower bound often depend on the set size, i.e., they must be computed separately for each (pair of) set(s) [81].

SIGNATURES FOR SETS Various signature schemes for sets have been proposed. An inverted index based on a particular signature scheme *signame* is referred to as *signame*-based inverted index. For the discussion, we assume two sets r and s.

The *prefix* [6, 29] is a simple but effective signature [81]. To compute the prefix, the

Table 1.1: Similarity resp. distance functions for two sets, r and s, and a similarity resp. distance threshold t.

Function	Notation	Definition	Eq. Overlap	Min. Size	Norm.
Overlap	$O(r,s)$	$\lvert r \cap s \rvert$	t	t	×
Jaccard	$J(r,s)$	$\frac{\lvert r \cap s \rvert}{\lvert r \cup s \rvert}$	$\frac{t}{1+t}(\lvert r \rvert + \lvert s \rvert)$	$t \cdot \lvert r \rvert$	✓
Cosine	$C(r,s)$	$\frac{\lvert r \cap s \rvert}{\sqrt{\lvert r \rvert \cdot \lvert s \rvert}}$	$t\sqrt{\lvert r \rvert \cdot \lvert s \rvert}$	$t^2 \cdot \lvert r \rvert$	✓
Dice	$D(r,s)$	$\frac{2\lvert r \cap s \rvert}{\lvert r \rvert + \lvert s \rvert}$	$\frac{t(\lvert r \rvert + \lvert s \rvert)}{2}$	$\frac{t\lvert r \rvert}{2-t}$	✓
Hamming	$H(r,s)$	$\lvert (r \cup s) \setminus (r \cap s) \rvert$	$\frac{\lvert r \rvert + \lvert s \rvert - t}{2}$	$\lvert r \rvert - t$	×

tokens of all sets must be sorted according to a total token order. The prefix signature considers the first π_r tokens of a set r, and each individual token is a signature of r. An effective heuristic is to order the tokens in ascending global token frequency [81, 130], i.e., infrequent tokens occur before frequent tokens. Consequently, the prefix contains infrequent tokens. Intuitively, this results in fewer candidates because the chance of two sets sharing infrequent tokens is low. If the prefixes of two sets r and s do not share any token, then r and s cannot be similar. The length of the prefix, π_r, is chosen such that the remaining non-prefix tokens cannot satisfy the given threshold t even if they are identical. The prefix length typically depends on the set sizes and the equivalent overlap of the similarity function [81, 130]. Since the equivalent overlap may vary for each pair (r, s), the prefix length may also vary for each pair of sets.

Figure 1.7 illustrates the prefix filtering principle for two sets, r and s, and a prefix length of 3 (which corresponds to an overlap threshold of $t = 4$, i.e., r and s are similar if and only if $|r \cap s| \geq t = 4$). The prefixes of r and s are highlighted in red and blue, respectively. Assume that the non-prefix tokens are unknown, denoted "?". Due to the global token order, we know that any unknown token in r is ≥ 9 and any unknown token in s is ≥ 7. The maximum overlap is achieved if all unknown tokens of s find a match in r, i.e., the maximum overlap is 3. Consequently, r and s are guaranteed to be dissimilar since the prefixes share no token. We refer to Xiao et al. [130] for a detailed discussion on prefixes and optimizations.

$t = 4$ $|r| = |s| = 6$ Numerical token order: 1, 2, ...

r: 1 4 8 ? ? ?   s: 3 5 6 ? ? ?

Figure 1.7: Prefix filtering exemplified for two sets, r and s, and a prefix length of 3.

Another signature scheme partitions the tokens into non-overlapping subsets [37, 108], each of which is a signature of r. We refer to signatures generated by this scheme as *partition-based* signatures. In a nutshell, this signature scheme uses a hash function to map each individual token of r to a partition, and the tokens in a partition form a signature (in practice, the partition identifier is used as signature). If two sets r and s are similar, they must share at least one signature. The given threshold t is typically converted into an equivalent Hamming distance t_H, and every signature of r that is not in s (and vice versa) increases the Hamming distance of r and s by one. Consequently,

if this count exceeds t_H, then r and s are dissimilar. The number of partitions and the partition strategy depend on the set sizes and the similarity function [37]. Along similar lines, k-wise signatures [123] that are based on a combination of tokens have been proposed.

Finally, the signature scheme called *CoveringLSH* [86, 87, 93] is tailored to the Hamming distance and was initially proposed for binary vectors. Interestingly, it can be adapted to work for sets, which is beneficial if the binary vectors are sparse. The main advantage of CoveringLSH is that it produces significantly fewer candidates for some datasets as compared to the other signatures presented in this thesis. However, this comes at the cost of a large memory footprint.

Chapter 3 of this thesis presents an efficient solution to compute the density-based clustering for collections of sets. Our solution is evaluated using a prefix-based inverted index and the Hamming distance. However, it generalizes to other set index structures and any of the mentioned set similarity and set distance functions. Chapter 4 presents an extension for multi-core systems.

1.3.3 *Practical Use Case*

Various application areas for binary vector and set representations exist, for example, detecting joinable tables in data lakes [139], retrieving similar pairs of texts [114, 123], or online click fraud detection [83]. In this thesis, we discuss one application scenario from industry in more detail: trace clustering for process mining at Celonis SE. This application scenario was the main motivation for our density-based clustering solution presented in Chapter 3. Celonis SE is a software company based in Munich, Germany, that develops the market-leading software in the domain of process mining. Process mining aims to analyze and understand business processes based on event logs [1]. A *process* (or case) is represented as directed, timestamped graph of activities. In this graph, a node represents an activity a_i (i.e., an atomic part of the process) and an edge (a_i, a_j) from activity a_i to activity a_j implies that a_j follows a_i. To assess the similarity of two processes, each distinct edge (a_i, a_j) gets a unique identifier, and a process is stored as a multiset of edge identifiers [1].

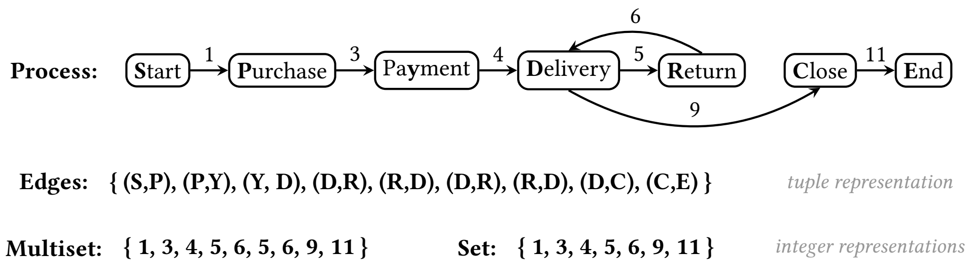


Figure 1.8: Example process and its representations.

Figure 1.8 shows an example process with seven activities: **S**tart, **P**urchase, Pa**y**ment, **D**elivery, **R**eturn, **C**lose, and **E**nd. The **S**tart and **E**nd activities exist in every process, and a process may contain loops. For example, a wrong or damaged order may be

returned and the replacement product is delivered (again), which results in a loop between **D**elivery and **R**eturn. Typically, each distinct activity tuple is assigned a unique integer, which enables efficient comparison of edges. This is shown on the bottom of Figure 1.8. Since an activity pair may appear multiple times in a process, the resulting representation is a multiset. Whether multisets are transformed into sets depends on the application. In our particular use case, we assume sets (i.e., tokens are deduplicated).

A company may store distinct processes with identical sequences of activities. We keep only a single representative of such processes, the so-called *trace*. The collection of traces then may serve as an input to queries, for example, density-based clustering (cf. Chapter 3).

1.4 CONTRIBUTIONS

Efficient and scalable similarity queries are at the core of this thesis. In particular, this thesis focuses on two specific types of similarity queries: (1) Top-k subtree similarity queries for trees and (2) density-based clustering for sets. For both query types, our solutions have been published as peer-reviewed conference papers.

In addition to the technical and algorithmic contributions, reproducibility played an important role for this work and resulted in the following artifacts: (i) A journal article on data reproducibility (in collaboration with other members of the Database Research Group at the University of Salzburg; not included in this thesis). (ii) A reproducibility package of our solution to answer top-k subtree similarity queries. The contributions can be summarized as follows.

1.4.1 *Scalable Top-k Subtree Similarity Queries*

The efficient retrieval of the k most similar subtrees in a large document tree with respect to a given query tree constitutes an important query type. This is referred to as *top-k subtree similarity query*, where the trees are typically compared using the tree edit distance. Previous solutions do either not use an index (and must therefore scan the entire dataset) or build an index that is quadratic in the input size.

We developed a novel solution for this problem that is based on a linear-space index structure. The index structure is organized as an inverted list index, but avoids full materialization of the inverted lists in main memory (which would require quadratic space). Instead, relevant parts of the full lists are built on the fly. Our clever traversal of the index structure, called *candidate score order*, processes the most promising subtrees first. As a consequence, our solution finds the final result with a small number of tree edit distance computations. This results in runtime improvements of up to four orders of magnitude compared to the state of the art. Finally, our index is the first incrementally updatable, linear-space index structure for top-k subtree similarity queries.

1.4.2 *Density-Based Clustering for Sets*

Density-based clustering is a widely used clustering technique with the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm being the most popular representative. DBSCAN is able to identify clusters of arbitrary shape, which are dense regions that are separated by regions of low density. Clusters are formed by recursively expanding dense neighborhoods until a low-density neighborhood is reached (i.e., the number of neighbors falls below a given threshold). Thus, DBSCAN imposes a partial order on the neighborhood computations and requires so-called *symmetric* index structures to retrieve *all* neighbors of a particular data point. Unfortunately, symmetric index structures are inefficient compared to their asymmetric counterparts.

To the best of our knowledge, we developed the first DBSCAN-compliant algorithm that is able to use asymmetric indexes. Asymmetric indexes rely on a particular processing order and retrieve only a specific part of all neighbors. Our solution imposes a processing order that is compatible with asymmetric indexes and produces a DBSCAN-compliant clustering. We compare our solution to a join-based approach, which also uses asymmetric indexes but needs to materialize the neighborhoods in main memory. In the worst case, the neighborhood materialization requires quadratic memory. In contrast, our solution runs in linear space while exploiting the effectiveness of asymmetric indexes. Our experiments suggest that our solution combines the best of two worlds: it is competitive with the join-based solution in terms of runtime performance and retains the memory efficiency of the DBSCAN algorithm with symmetric index.

1.4.3 *Reproducibility*

Reproducibility of data and experiments constitutes an important part in computer science research. Without reproducibility, other research groups may be unable to reproduce previous experiments and this may prevent or delay future research. Therefore, the Database Research Group at the University of Salzburg published a collaborative work on data reproducibility, and a reproducibility package for our ACM SIGMOD 2019 paper (cf. Chapter 2) was created.

DATA REPRODUCIBILITY Pawlik, Hütter, Kocher, Mann, and Augsten. A Link is not Enough - Reproducibility of Data. In *Datenbank Spektrum 19*, pages 107–115, June 2019. Springer.

Pawlik et al. [92] is a collaborative work of the Database Research Group at the University of Salzburg on the problem of data reproducibility. In this work, we introduce the RPI data reproducibility model that covers three elements of data reproducibility: Given the *raw data* (R), *preparation instructions* (P) are used to derive the *input data* (I) for an experimental evaluation. Only providing access to the raw data is not enough and often prevents experiments from being reproduced. This work also consists of an extensive review of related work and reproducibility efforts in the database community,

legal and technical aspects of data availability, and best practice examples. Due to its collaborative nature, this journal article is not part of this thesis.

REPRODUCIBILITY PACKAGE The ACM SIGMOD conference has initiated a program to promote reproducible papers[2]. Our reproducibility package has been accepted to the ACM SIGMOD 2020 Reproducibility program. It executes all experiments of our paper *A Scalable Index for Top-k Subtree Similarity Queries* [69] (cf. Chapter 2) and recompiles the paper sources with the new results. The report of our reproducibility package is included in Appendix A.

1.5 THESIS OUTLINE

This thesis is organized as a collection of conference papers. Chapters 2 and 3 are self-contained, and all experimental results can be found in the respective chapters. Chapter 4 extends the results of Chapter 3 to multi-core environments and has not yet been published. The bibliography for all chapters is provided at the end of this thesis.

CHAPTER 2 A Scalable Index for Top-*k* Subtree Similarity Queries

> Daniel Kocher and Nikolaus Augsten. A Scalable Index for Top-*k* Subtree Similarity Queries. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 1624–1641, Amsterdam, Netherlands, July 2019. ACM.

CHAPTER 3 Scaling Density-Based Clustering to Large Collections of Sets

> Daniel Kocher, Nikolaus Augsten, and Willi Mann. Scaling Density-Based Clustering to Large Collections of Sets. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021*, Nicosia, Cyprus, March 2021. OpenProceedings.org. Accepted.

CHAPTER 4 A Multi-Core Solution for Density-Based Clustering of Sets

> This chapter extends the results of Chapter 3 to multi-core environments and includes algorithms as well as experimental results.

2 https://reproducibility.sigmod.org/

2

A SCALABLE INDEX FOR TOP-K SUBTREE SIMILARITY QUERIES

AUTHORS Daniel Kocher and Nikolaus Augsten
VENUE ACM SIGMOD, Amsterdam, 2019

ABSTRACT

Given a query tree Q, the *top-k subtree similarity query* retrieves the k subtrees in a large document tree T that are closest to Q in terms of tree edit distance. The classical solution scans the entire document, which is slow. The state-of-the-art approach precomputes an index to reduce the query time. However, the index is large (quadratic in the document size), building the index is expensive, updates are not supported, and data-specific tuning is required.

We present a scalable solution for the top-k subtree similarity problem that does not assume specific data types, nor does it require any tuning. The key idea is to process promising subtrees first. A subtree is promising if it shares many labels with the query. We develop a new technique based on inverted lists that efficiently retrieves subtrees in the required order and supports incremental updates of the document. To achieve linear space, we avoid full list materialization but build relevant parts of a list on the fly.

In an extensive empirical evaluation on synthetic and real-world data, our technique consistently outperforms the state-of-the-art index w.r.t. memory usage, indexing time, and the number of candidates that must be verified. In terms of query time, we clearly outperform the state of the art and achieve runtime improvements of up to four orders of magnitude.

2.1 INTRODUCTION

Data with hierarchical structure are naturally represented as trees. A tree stores data values in node labels and encodes the relation between the values in the structure (e.g., text values and element nesting in XML). We consider applications that, given an example tree (the query), are interested in subtrees of a large document tree that are similar to the query. An example is the abstract syntax tree of a large software project [44, 102]: In order to avoid code duplication or detect code moves, software engineers are interested in finding all code fragments (i.e., subtrees of the abstract syntax tree) that are similar to a given example fragment. In RNA secondary structures (which are represented as ordered, labeled trees [3, 56]), biologists search for similar foldings of RNA subsequences. To automatically extract product information from the web, the similarity of substructures in web pages are leveraged [119]. Production

engineers retrieve components with similar building plans from bills of materials, which form trees that may consists of millions of nodes [46, 65].

We study *top-k subtree similarity queries*: given a large document tree T and a (small) query tree Q, find the k most similar subtrees in T w.r.t. Q. Two trees are similar if their *edit distance* [136], a common tree similarity measure, is small. The edit distance between two ordered labeled trees is defined as the minimum number of node edit operations (insertion, deletion, rename) that transform one tree into the other.

Previous solutions for top-k subtree similarity queries fall into two categories: index-based and index-free algorithms. TASM-Postorder [8] is the fastest index-free algorithm and runs in small memory. Unfortunately, TASM-Postorder must scan the entire document to answer a top-k query, which is slow. StructureSearch [31] addresses this issue and leverages a precomputed index to retrieve candidate subtrees. The candidates must be verified using the edit distance.

StructureSearch runs faster than TASM-Postorder but suffers from the following issues: (1) The index size is quadratic in the document size n for deep trees; note that the document is the database over which we answer the top-k query. (2) Despite the index, StructureSearch must retrieve and verify many subtrees, which leads to high runtimes also for small values of k. (3) While StructureSearch can be generalized to generic tree data, the solution is tailored to XML documents, which have many repeating labels in the inner nodes (element tags) and infrequent labels in the leaves (text values). Further, XML trees are typically flat. Flat trees are in favor of StructureSearch since the index grows larger for deep trees. (4) The index is not updatable.

Our solution is based on the idea of a *candidate score*. The candidate score ranks all subtrees of a document. The score is high if the query and the subtree share many labels. Intuitively, subtrees with a high score are more likely to be close to the query in terms of edit distance. By processing subtrees in candidate score order, we (a) find good candidates quickly and (b) can stop early when the ranking is good enough, i.e., all remaining subtrees cannot improve the ranking. Stopping early is possible since the candidate score implies a lower bound on the edit distance. The candidate score is very effective. In many settings, we verify orders of magnitude fewer candidate subtrees than StructureSearch; in some settings we only verify k candidates, which is optimal.

The challenge is to efficiently generate candidates in score order. The query is not known upfront, thus the order must be established at query time and cannot be precomputed. It is clearly not feasible to enumerate all subtrees and sort them by their score. We introduce a new technique that is based on an inverted list index over the document node labels. The inverted list of a label stores all subtrees that contain that label. We split the lists into partitions of subtrees with the same size and show how to leverage the list partitions to processes the subtrees in score order. The partitions are accessed in the order of the best candidate score that may be found in that partition. Only relevant partitions need to be accessed, e.g., there is only a single partition that may contain subtrees of the highest score.

The catch is that the label inverted list index is quadratic in the document size n for tress with depth $O(n)$; for such trees, also the index of the state-of-the-art algorithm, StructureSearch, is quadratic. We propose a new algorithm, SlimCone, which uses an

incrementally updatable, linear-space index structure to build the relevant partitions of the inverted lists on the fly at query time. SlimCone verifies the subtrees in non-decreasing candidate score order. We show how to generate partitions efficiently such that the performance penalty of generating the partitions on the fly is small.

Summarizing, our contributions are the following.

- We propose SlimCone, a new, index-based algorithm for the top-k subtree similarity problem. SlimCone verifies subtrees in decreasing candidate score order, i.e., more promising subtrees are processed first. SlimCone does not require any parameters and is not tailored to a specific data type.
- The state-of-the-art algorithm uses a quadratic-size index. We propose the first linear-space index for top-k subtree similarity queries. Our index groups subtrees into partitions. All subtrees in a partition have the same guarantee w.r.t. to the candidate score such that we find promising subtrees efficiently.
- We propose an extension of SlimCone that supports incremental index updates. Previous work must recompute the index from scratch when the document tree is updated.
- We empirically evaluate our solution on large synthetic and real-world datasets. Our technique clearly outperforms the state of the art w.r.t. memory usage, indexing time, number of verified candidates, and query runtime, often by orders of magnitude.

The remaining chapter is organized as follows. Section 2.2 provides background material and introduces the problem statement. Section 2.3 discusses the candidate scores. In Sections 2.4-2.6 we present our index structures and algorithms[1]. Section 2.7 describes how to make our index incrementally updatable. We discuss related work in Section 2.8. Before we conclude in Section 2.10, we provide empirical evidence of the scalability and efficiency of our solution in Section 2.9. Appendix A of this thesis contains the report of our reproducibility package, which has been accepted to the ACM SIGMOD 2020 Reproducibility program.

2.2 NOTATION, BACKGROUND, AND PROBLEM STATEMENT

TREES We assume *rooted, ordered, labeled trees.* A *tree* T is a directed, acyclic, connected graph with nodes $V(T)$ and directed edges $E(T) \subseteq V(T) \times V(T)$. Each node has at most one incoming edge, the node with no incoming edge is the *root node*. The size of a tree, $|T| = |V(T)|$, is the number of its nodes. In an edge $(u, v) \in E(T)$, u is the *parent* of v, denoted $par(v)$, and v is the *child* of u. Two nodes are *siblings* if they have the same parent. A *leaf node* has no children. Each node u has a *label*, $\lambda(u)$, which is not necessarily unique. The multiset of all labels in T is $\mathcal{L}(T)$. The *postorder* (*preorder*) *identifier* of node u, $post(u)$ ($pre(u)$), is the postorder (preorder) position of u in the

1 The given proofs and pseudocodes in these sections can be found in the Appendix of the original publication, Kocher and Augsten [69].

tree (1-based numbering). The trees are *ordered*, i.e., the sibling order matters. If node u is on the path from the root to node v, $u \neq v$, then v is a *descendant* of u, and u is a *ancestor* of v. A *subtree* T_u of T is a tree that consists of node u, all descendants of u, and all edges in $E(T)$ connecting these nodes.

TREE EDIT DISTANCE The edit distance, $\delta(S,T)$, between two trees, S, T, is the minimum number of node edit operations that transforms S into T. We assume the standard node operations [136]: *Rename* changes the label of a node. *Delete* removes a node u and connects the children of the deleted node to its parent, starting at the sibling position of u and maintaining the sibling order. *Insert* adds a new node u as the i-th child of an existing node p, replacing a (possibly empty) sequence $C = (c_i, c_{i+1}, \ldots, c_j)$ of p's children; the child sequence C is connected under the new node u. Insert and delete are reverse operations. The fastest algorithms for the tree edit distance run in $O\left(|T|^3\right)$ time and $O\left(|T|^2\right)$ space [91], i.e., computing the edit distance is expensive and should be avoided.

LOWER BOUNDS A lower bound for the tree edit distance may underestimate the true distance, but never overestimates it. A number of edit distance lower bounds have been defined [76]. Lower bounds are typically computed much faster than the edit distance. We leverage the *label lower bound*,

$$llb(S,T) = max\{|S|, |T|\} - |\mathcal{L}(S) \Cap \mathcal{L}(T)| \leq \delta(S,T), \tag{2.1}$$

where $A \Cap B$ denotes the intersection between two multisets, A and B, and the *size lower bound*,

$$slb(S,T) = ||S| - |T|| \leq \delta(S,T) \tag{2.2}$$

Definition 2.2.1 (Top-k Subtree Similarity Query)**.** *Given a query tree Q, a document tree T, $k \leq |T|$. The* top-k subtree similarity query *returns a top-k ranking R, where R is the sequence of the k most similar subtrees of document T w.r.t. query Q such that $\forall T_j \notin R, T_i \in R.\, \delta(Q,T_i) \leq \delta\left(Q,T_j\right)$. The subtrees in $R = [T^1, T^2, \ldots, T^k]$ are sorted by their edit distance to Q, i.e., $\forall 1 \leq i < j \leq k.\, \delta\left(Q,T^i\right) \leq \delta\left(Q,T^j\right)$.*

PROBLEM STATEMENT Our goal is a time- and space-efficient solution for the top-k subtree similarity query that scales to large document trees.

A naive solution computes the edit distance $\delta(Q,T_i)$ for all subtrees $T_i \in T$, sorts them by $\delta(Q,T_i)$, and returns the first k subtrees in ascending sort order. Obviously, this approach does not scale to large documents [8]. Efficient techniques prune irrelevant subtrees and compute the edit distance only for candidate subtrees that cannot be filtered. Well known filter techniques include the following.

SIZE FILTER Augsten et al. [8] show that only subtrees of a maximum size $\tau = 2|Q| + k$ need to be considered, thus subtrees T_i, $|T_i| > \tau$, can be pruned.

RANKING FILTER Once an intermediate ranking R' of size k is obtained, the edit distance $\delta(Q, R'[k])$ ($\delta(R'[k])$ for short) between the query Q and the last tree $R'[k]$ in the ranking serves as a filter: A subtree $T_i \notin R'$ improves the final ranking R' iff $\delta(Q, T_i) < \delta(R'[k])$ [8]. Together with a lower bound, $lb(Q, T_i)$, a subtree can be safely pruned if $lb(Q, T_i) \geq \delta(R'[k])$.

The better the ranking, the more effective is the ranking filter. Thus, to reduce the number of verifications it is important to find good subtrees early in the process.

Table 2.1 provides an overview of our notation.

Table 2.1: Notation overview.

Notation	Description
T/Q	document / query tree
R/R'	final / intermediate top-k ranking
k	results size, $k = \|R\|$
$R[j]$	j-th entry in R
T_i	a subtree $T_i \in T$
$par(u)$	parent of node u
$pre(u)/post(u)$	preorder / postorder identifier of node u
$\lambda(u)$	label of node u
$\mathcal{L}(T_i)$	label multiset of tree T_i
$\delta(Q, T_i)$	edit distance btw. Q and T_i
$\delta(R[j])$	edit distance btw. Q and j-th entry in R
$slb(Q, T_i)$	size lower bound btw. Q and T_i
$llb(Q, T_i)$	label lower bound btw. Q and T_i
τ $(= 2\|Q\| + k)$	maximum relevant subtree size [8]

2.3 EFFECTIVE CANDIDATE GENERATION

The key idea of our approach is to prioritize promising subtrees. If we fill the ranking with good subtrees, the ranking filter (cf. Section 2.2) is effective and we can terminate early. In this section we define the *candidate score* to rank subtrees. In the following sections we discuss how to retrieve subtrees in the order of their candidate score.

Definition 2.3.1 (Candidate Score)**.** *Given query Q and document T, the candidate score of a subtree T_i of T is*

$$score(T_i) = \frac{1}{1 + llb(Q, T_i)},$$

where $llb(Q, T_i)$ is the label lower bound between Q and T_i.

The candidate score is in the interval $(0, 1]$, more promising subtrees score higher. The candidate score imposes a total order on the subtrees of document T, which we call *candidate score order*: Given two subtrees $T_i, T_j \in T$, $T_i > T_j$ iff $score(T_i) > score(T_j)$.

A subtree T_i is processed by computing the tree edit distance between T_i and the query Q, and by inserting T_i into the ranking if $\delta(Q, T_i) < \delta(R[k])$. If we process the

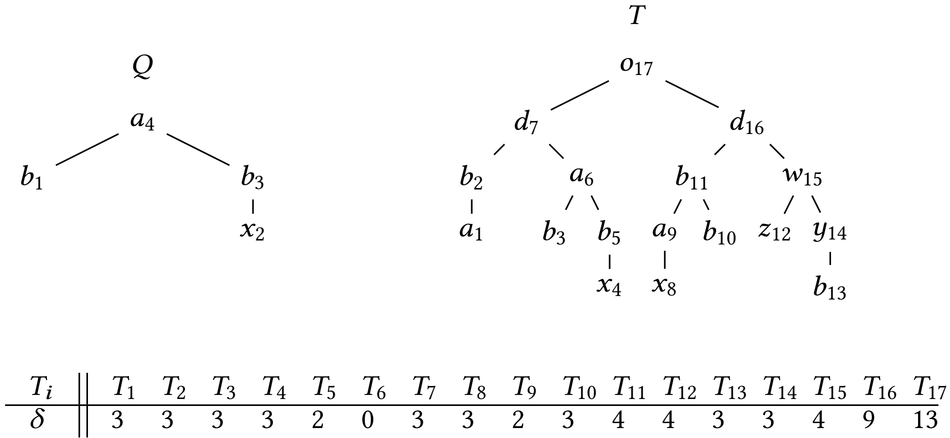
subtrees in candidate score order, we can stop after m subtrees if the following stopping condition holds.

Lemma 2.3.2 (Early Termination)**.** *Let T^i be the i-th subtree of document T in candidate score order w.r.t. query Q (breaking ties arbitrarily), R' a top-k ranking of the subtrees $T^1, T^2, \dots, T^m$, $k \leq m < |T|$. If $\delta(R'[k]) \leq llb\left(Q, T^{m+1}\right)$, then R' is a valid top-k ranking for all subtrees $T^i \in T$.*

Proof. Due to the candidate score order and Def. 2.3.1, $\delta(R'[k]) \leq llb\left(Q, T^j\right)$ for all $j > m$; since $llb\left(Q, T^j\right) \leq \delta\left(Q, T^j\right)$, no subtrees T^j can improve the ranking. □

SIMPLE ALGORITHM A simple top-k subtree similarity algorithm, SIMPLE, that uses Lemma 2.3.2 and the size filter (cf. Section 2.2) proceeds as follows: compute the score for each subtree $T_i \in T$, $1 \leq |T_i| \leq \tau$, and sort all subtrees by score, process the subtrees in sort order, and stop when the early termination condition holds.

RUNNING EXAMPLE Figure 2.1 shows an example document T, an example query Q, and the edit distance (δ) for all subtrees $T_i \in T$ w.r.t. Q. Each node is represented by its label and the postorder identifier (subscript number). In the examples, we refer to the subtree rooted in the i-th node of T in postorder as T_i.



T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}
δ	3	3	3	3	2	0	3	3	2	3	4	4	3	3	4	9	13

Figure 2.1: Running example.

Example 2.3.3. *We compute the top-k ranking, $k = 3$, for Q in T using SIMPLE (cf. Figure 2.1). Due to the size filter, the maximum subtree size that must be considered is $\tau = 2|Q| + k = 11$. We compute the label lower bound for all subtrees T_i, $|T_i| \leq \tau$ and rank them by candidate score. For example, the label lower bound for T_9 is $llb(Q, T_9) = max\{|Q|, |T_9|\} - |\mathcal{L}(Q) \Cap \mathcal{L}(T_9)| = 2$, where $\mathcal{L}(Q) = \{\{a, b, b, x\}\}$ and $\mathcal{L}(T_9) = \{\{a, x\}\}$; the candidate score of T_9 is $score(T_9) = 1/3$. The result is shown in Table 2.2; we order subtrees by postorder position in the case of ties; T_{17} is not listed since $|T_{17}| > \tau$.*

SIMPLE first processes T_6, T_{11}, and T_2 in this order and computes $\delta(Q, T_6) = 0$, $\delta(Q, T_{11}) = 4$, and $\delta(Q, T_2) = 3$, resulting in the intermediate ranking $R' = [T_6, T_2, T_{11}]$. Since

Table 2.2: Example subtrees ordered by candidate score.

$llb(Q, T_i)$	$score(T_i)$	Subtrees
0	1	T_6, T_{11}
2	1/3	T_2, T_5, T_9
3	1/4	T_1, T_3, T_4, T_7, T_8, T_{10}, T_{13}, T_{14}, T_{15}
4	1/5	T_{12}
5	1/6	T_{16}

$\delta(R'[k]) = 4$ and $llb(Q, T') = 2$ for the next unprocessed subtree $T' = T_5$, we continue and verify T_5 and T_9. $\delta(Q, T_5) = 2$, $\delta(Q, T_9) = 2$, resulting in $R' = [T_6, T_5, T_9]$. Now, $\delta(R'[k]) = 2 \leq llb(Q, T') = 3$ for the next subtree $T' = T_1$, and we can terminate.

2.4 INDEX AND MERGEALL ALGORITHM

We introduce the *candidate index*, which enables us to efficiently retrieve candidates in score order, and propose MergeAll, a baseline algorithm that solves top-k subtree similarity queries using our index.

2.4.1 *Candidate Index*

The candidate index, $\mathcal{I}$, is built over a document tree, T, and stores the following data structures:

1. An *inverted list index* over the document labels.

2. The *node index*, a compact representation of T.

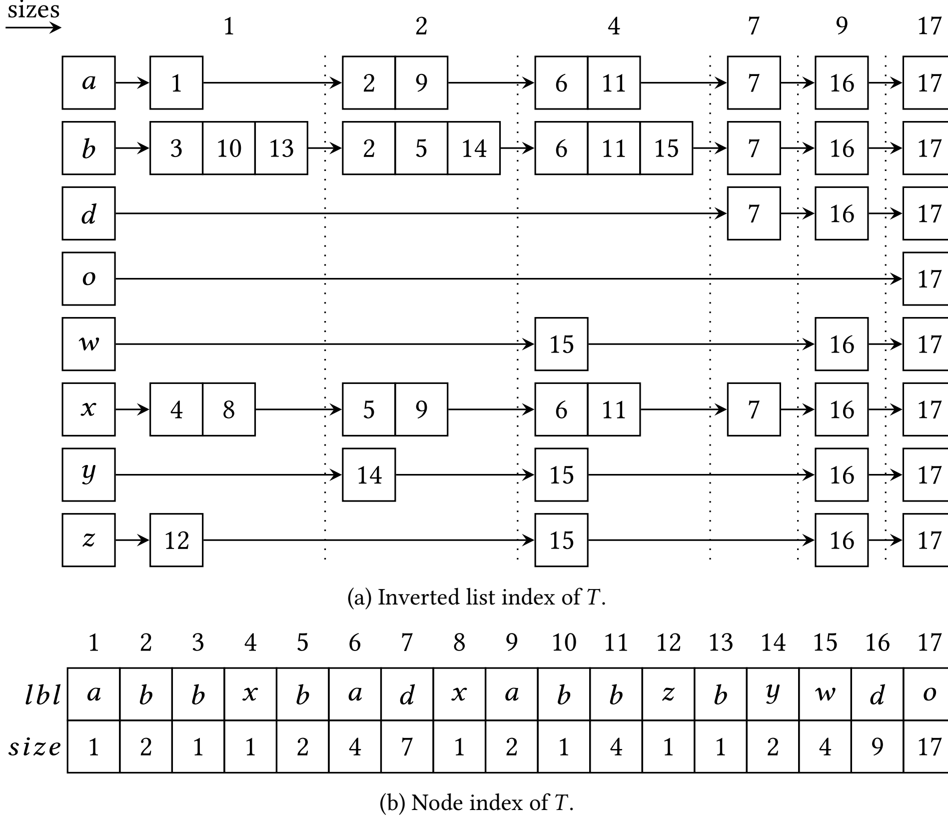
Our index supports the following operations:

- $\mathcal{I}.list(\lambda)$ retrieves the inverted list l_λ for a label λ and returns *nil* if that list does not exist.

- $\mathcal{I}.sizes()$ retrieves all distinct subtree sizes in T.

INVERTED LIST INDEX We build an inverted list index on the document labels. For each *distinct* label $\lambda \in \mathcal{L}(T)$, we maintain a list l_λ of all subtrees that contain a node labeled λ. The inverted list entries are lexicographically sorted by subtree size and postorder identifier (ascending order). Figure 2.2a shows the inverted lists for our example document. A list entry is a subtree T_i, represented by the postorder identifier of its root node, i. The lists are partitioned by subtree sizes (shown above the lists).

NODE INDEX We store the document T in an array of size $|T|$. The i-th field in the array (1-based counting) is a pair $(\lambda_i, |T_i|)$, where λ_i is the label of the i-th node of T in postorder and $|T_i|$ is the size of the subtree rooted in that node. Figure 2.2b shows the node index for our example document.



(a) Inverted list index of T.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lbl	*a*	*b*	*b*	*x*	*b*	*a*	*d*	*x*	*a*	*b*	*b*	*z*	*b*	*y*	*w*	*d*	*o*
size	1	2	1	1	2	4	7	1	2	1	4	1	1	2	4	9	17

(b) Node index of T.

Figure 2.2: Baseline index structure for document T of our running example (cf. Figure 2.1).

The node index is a lossless and compact representation of the document tree. We do not need any other representation of the document for our algorithms. Conveniently, each subtree T_i in the node index is a connected subsequence starting at position $i - |T_i| + 1$ ($|T_i|$ is accessed in constant time in the node index) and ending at position i. The subtree part of the node index is a valid tree representation by itself.

The node index is built in a single scan of the document using a SAX parser and is stored in main memory. While parsing, we build a dictionary that maps string labels to unique integers. In our indexes and algorithms (including the edit distance computation), we use integer labels (in our examples, however, we show the original string labels).

$$\lambda_1 \text{ ——— } \lambda_2 \text{ ——— } \lambda_3 \text{ ——— } \cdots \text{ ——— } \lambda_{n-1} \text{ ——— } \lambda_n$$

Figure 2.3: Worst-case document for the inverted list index (root to the left, leaf to the right).

INDEX SIZE The size of the candidate index is $O(n^2)$, $n = |T|$. Consider the tree in Figure 2.3 with root label λ_1, a single leaf λ_n, and pairwise distinct labels, $\lambda_i \neq \lambda_j$ unless $i = j$. The inverted list of λ_i has i entries, e.g., λ_n appears in n subtrees. The overall number of entries is $\sum_{i=1}^{n} i$, which is quadratic.

For the tree in Figure 2.3, also the index of StructureSearch [31] requires quadratic space. We introduce a linear-space index in Section 2.6.

2.4.2 *MergeAll Algorithm*

MergeAll uses the candidate index and processes the subtrees T_i in the order of non-decreasing size lower bound, $slb(T_i, Q) = ||T_i| - |Q||$ (cf. Section 2.2), with respect to the query Q.

We (conceptually) split the inverted lists into vertical *stripes* as illustrated in Figure 2.4. A stripe S_j consists of all subtrees T_i in all lists that have size $|T_i| = |Q| + j$, e.g., S_2 and S_{-2} are the blue stripes in the figure. A *partition* consists of all subtrees of a stripe in a single inverted list, e.g., the subtrees in stripe S_0 of list l_{λ_4} form a partition (marked in the figure). Stripes and partitions may be empty.

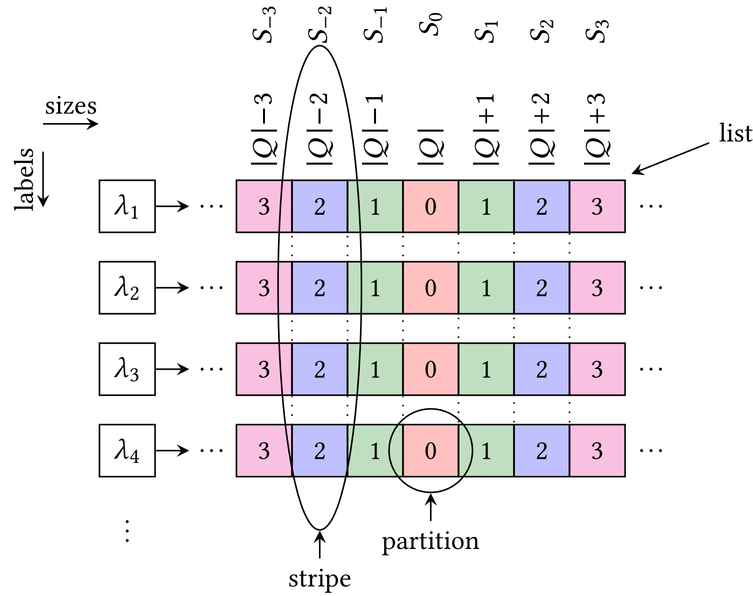


Figure 2.4: Stripes and partitions w.r.t. query Q.

OVERVIEW MergeAll processes the subtrees stripe by stripe. The current stripe number is j. We leverage the fact that the size lower bound for all subtrees T_i in S_j and S_{-j} is $slb(T_i, Q) = j$. By incrementing the stripe number we process the subtrees in ascending size lower bound order.

The goal, however, is to retrieve the subtrees in non-increasing candidate score order, which is equivalent to the non-decreasing label lower bound order. We maintain a *lower bound cache* (*lbc*) that stores subtrees in buckets. A subtree T_i in stripe S_j or S_{-j} with label lower bound $lb = llb(Q, T_i)$ is cached in bucket $lbc[lb]$ for later verification if $lb > j$.

We only process lists of labels that exist in Q, $\lambda \in \mathcal{L}(Q)$, therefore we have at most $|Q|$ lists. We start at stripe $j = 0$ and proceed in four steps:

1. Verify all subtrees in lower bound bucket $lbc[j]$.
2. For each candidate $T_i \in S_j \cup S_{-j}$ compute $lb = llb(Q, T_i)$.
   a) If $lb = j$, then verify T_i;
   b) otherwise, cache T_i in lower bound bucket $lbc[lb]$.
3. Increment to next stripe: $j \leftarrow j + 1$
4. Continue at step (1).

Whenever we verify a subtree T_i, we also update the ranking R. Since the current stripe number j is a size lower bound for all subtrees in S_j, we can terminate if $|R| = k$ and $j \geq \delta(R[k])$.

Overlap computation. We maintain two pointers, l and r, in each list. r is initialized to the first subtree T_i (subtree with the smallest postorder identifier) of stripe S_0, l starts at position $r - 1$. If not clear from the context, we refer to the pointers of a list l_λ by $l_\lambda.l$ and $l_\lambda.r$.

We move the pointers in an n-way merge fashion to compute the label overlap with the query. We stop moving a pointer when it points to the next stripe. We first move the l pointers and maintain a counter $ol[T_i]$ for each subtree T_i that we encounter; then we move the r pointers in a similar way. After all pointers stop, the counter $ol[T_i]$ stores the overlap $|\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$. This works because our index structure sorts elements within a stripe consistently. With the overlap, we compute the label lower bound, $llb(Q, T_i) = max\{|Q|, |T_i|\} - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)| \leq \delta(Q, T_i)$.

We next discuss two special cases. (1) *Duplicate query labels.* When the query Q has duplicate labels, the list l_λ is retrieved x times if Q has x nodes with label λ. Then, for a subtree T_i we get an overlap $ol[T_i] > |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$ if T_i has fewer than x nodes with label λ. The top-k result is still correct, but T_i may be processed too early w.r.t. to the candidate score order. To avoid this situation, we can collect all subtrees and compute their label overlap using our node index. In practice, the small violations of the candidate order have little effect, and we suggest using the merge approach. (2) *Lists without query label.* After processing all lists of the labels in Q, one of the following situations may happen. (a) $|R| < k$, i.e., we did not find k subtrees that have a common label with Q; (b) $\delta(R[k]) > |Q|$, i.e., there may be subtrees that do not share a label with Q but should be in the ranking. In this case, we need to consider lists of labels that do not exist in Q. For all subtrees in lists of non-query labels the minimum edit distance is $|Q|$. We merge the lists stripe by stripe and use the stopping condition to terminate. This corner case rarely appears in practice.

The following theorem considers MergeAll with the fix for duplicate query labels.

Theorem 2.4.1. *MergeAll solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Proof. Correctness: The stopping condition, $|R| = k \wedge j \geq \delta(R[k])$, is correct since all subtrees T_i in partitions that are not processed have size lower bound $slb(Q, T_i) \geq j$ and can therefore not improve the ranking. *Candidate score order*: We increment stripe number j, starting with $j = 0$. For a given j, we perform two steps: (1) We postpone the verification of subtrees $T_i \in S_j \cup S_{-j}$ for which $x = llb(Q, T_i) > j$ and cache them in $lbc[x]$. (2) We verify (a) all subtrees $T_i \in S_j \cup S_{-j}$ for which $llb(Q, T_i) = j$ *and* (b) all subtrees $T_i \in lbc[j]$ (cached subtrees from previous stripes, $j' < j$). Thus, all subtrees T_i of the stripes $j' < j$ with $llb(Q, T_i) = j$ are verified when we process stripe j. There exists no subtree T_i with $llb(Q, T_i) = j$ in some stripe $S_{j''}$, $j'' > j$, since $llb(Q, T_i) \geq slb(Q, T_i) = j'' > j$ for all subtrees in $S_{j''}$. □

Example 2.4.2. *Figure 2.5 illustrates MergeAll for our running example, $k = 3$. We retrieve the lists of the labels in Q: a, b (twice since b is a duplicate label), and x. We start with stripe S_0 (red stripe). Pointer r is initialized to the first subtree in S_0 (T_6 in all lists), l starts on the last subtree in the green partition. We compute the overlap by moving the pointers and merging the lists. l cannot be moved; r merges the partitions in S_0 and computes the overlaps of T_6 (4), T_{11} (4), and T_{15} (2). Note that the true overlap of T_{15} is 1; we overestimate due to the duplicate query label b. From the overlaps, we get the label lower bounds $llb(Q, T_6) = llb(Q, T_{11}) = 0$ and $llb(Q, T_{15}) = 2$. Hence, T_6 and T_{11} are verified, whereas T_{15} is cached in bucket $lbc[2]$; $R' = [T_6, T_{11}]$. For the next stripe, $j = 1$, there is nothing to do since $lbc[1]$, S_1, and S_{-1} are all empty. For $j = 2$, we first verify T_{15} in $lbc[2]$ and get the ranking $R' = [T_6, T_{11}, T_{15}]$; next we process the subtrees in stripe S_{-2} (green); S_2 is empty. The overlaps (2 for T_{14}, T_9, and 3 for T_5, T_2) are computed while l is decremented. T_{14}, T_9 are verified immediately. After T_5 is verified in the next round $j = 3$, $R = [T_6, T_5, T_9]$, and we terminate since $\delta(R[k]) \leq j$. Figure 2.5 illustrates the pointers after processing T_5.*

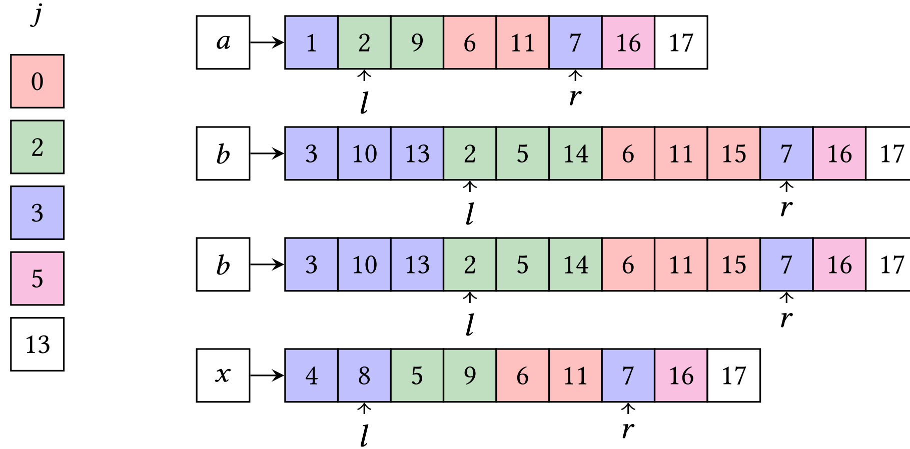


Figure 2.5: MergeAll after processing stripes $j = 2$.

PSEUDOCODE Algorithm 1 shows the pseudocode for MergeAll (Algorithms 2 and 3 are auxiliary functions).

```
Algorithm 1: MergeAll(Q, T, k)
   Input: Query tree Q, document tree T, result size k
   Result: Top-k ranking R of subtrees of T w.r.t. Q
   // I ... candidate index, L(Q) ... label multiset of Q
   // T_i ... subtree rooted at node i
1  foreach λ ∈ L(Q) do // initialize inverted lists
2      l_λ ← I.list(λ); // retrieve list l_λ
3      if l_λ ≠ nil then // initialize pointers l_λ.r, l_λ.l
4          l_λ.r ← pos. of i s.t. ||Q| − |T_i|| is minimal;
5          l_λ.l ← l_λ.r − 1
6  ol ← empty associative array; // overlap store
7  lbc ← empty dynamic array; // lower bound cache
8  j ← 0; // current stripe number
9  R ← empty ranking;
10 while j ≤ 2|Q| do // j > 2|Q|: we must consider all lists
11     if Verify-Bucket(j) then return R; // evaluate lbc[j]
12     foreach node i ∈ S_j ∪ S_−j do // compute overlaps
13         ol[i] ← # of lists l_λ s.t. i ∈ l_λ;
14         advance l_λ.r and l_λ.l;
15     foreach key i ∈ ol do // process subtrees (cache or verify)
16         lb ← max{|Q|, |T_i|} − ol[i];
17         if Process-Subtree(T_i, lb, j) then return R;
18     j ← j + 1; // proceed to next j' > j
       // check if we can terminate before continuing
19     if |R| = k ∧ j ≥ δ(R[k]) then return R;
20 return R;
```

```
Algorithm 2: Process-Subtree(T_i, lb, B)
   Input: Subtree T_i, lower bound lb, edit distance bound B ≤ lb
   Result: True if final ranking found, false otherwise
   // lbc, R, Q globally accessible
1  if lb > B then // cache T_i
2      lbc[lb] ← lbc[lb] ∪ {T_i};
3      return false; // we cannot terminate
4  compute δ(T_i, Q) and update R with T_i; // lb = B; verify T_i
5  return |R| = k ∧ B ≥ δ(R[k]); // indicates if we can terminate
```

```
Algorithm 3: Verify-Bucket(B)
   Input: Edit distance bound B
   Result: True if final ranking found, false otherwise
   // lbc, R, Q globally accessible
1  foreach T_i ∈ lbc[B] do // verify all subtrees in lbc[B]
2      compute δ(T_i, Q) and update R with T_i;
       // return as soon as we can terminate
3      if |R| = k ∧ B ≥ δ(R[k]) then return true;
4  return false; // we cannot terminate
```

2.5 CONE: PARTITION-BASED TRAVERSAL

MergeAll processes one stripe per round and computes the label lower bound for all subtrees in a stripe. The stripes may be large, leading to slow execution times. We observe, however, that the size of the partitions within a stripe may vary greatly. The inverted lists of frequent labels are very long (e.g., the list of the "article" tag in the DBLP bibliography), leading to large partitions. Then, the runtime is dominated by processing the partitions of long lists.

In this section we present Cone, an algorithm that addresses this issue. Cone processes only a subset of the partitions in each stripe. The inverted lists are sorted and short lists are accessed first. Therefore, the algorithm may terminate before considering any of the large partitions. Cone uses an edit distance bound $\mathcal{B}$, which is zero initially and is incremented in each round. Only partitions that possibly contain a subtree T_i at distance $\mathcal{B}$ from query Q are considered.

Assume we know that there are $nml(T_i)$ labels in Q that do not exist in subtree T_i. We call $nml(T_i) = |\mathcal{L}(Q) \setminus \mathcal{L}(T_i)|$ the *number of missing* labels in T_i w.r.t. Q. Then we can draw conclusions on the size of T_i that is required to achieve edit distance $\mathcal{B}$.

Theorem 2.5.1 (Size Interval)**.** *Let T_i be a subtree of document T, Q be the query tree, $nml(T_i)$ be the number of missing labels in T_i w.r.t. Q, and $\mathcal{B} \geq 0$ an edit distance bound. If $\delta(Q, T_i) \leq \mathcal{B}$, then $|T_i|$ is in the* size interval

$$si(\mathcal{B}, Q, T_i) = [|Q| - \mathcal{B}; |Q| + \mathcal{B} - nml(T_i)] \tag{2.3}$$

Proof. Recall that the number of missing labels in T_i w.r.t. Q is defined as $nml(T_i) = |\mathcal{L}(Q) \setminus \mathcal{L}(T_i)|$; $nml(T_i) \leq \mathcal{B}$ due to $\delta(Q, T_i) \leq \mathcal{B}$. We prove the correctness of the size interval by contradiction.

Case A: Assume a subtree T_i with $\delta(Q, T_i) \leq \mathcal{B}$ and $|T_i| \leq |Q| - \mathcal{B} - 1$. The minimum number of edit operations that transform any instance of T_i to some instance of Q consists of $|Q| - |T_i|$ insert operations. Note that we can decrease $nml(T_i)$ *and* the size difference $|Q| - |T_i|$ by inserting a new node with a label from $\mathcal{L}(Q) \setminus \mathcal{L}(T_i)$ into T_i. Thus, in the best case, we perform exactly $|Q| - |T_i|$ insertions, i.e., $\delta(Q, T_i) = |Q| - |T_i|$. Our assumption yields $|Q| - |T_i| \geq \mathcal{B} + 1$, hence $\delta(Q, T_i) \geq \mathcal{B} + 1$, which contradicts our assumption.

Case B: Assume a subtree T_i with $\delta(Q, T_i) \leq \mathcal{B}$ and $|T_i| \geq |Q| + \mathcal{B} - nml(T_i) + 1$. In this case, a delete operation can decrease the size difference $|T_i| - |Q|$ but cannot decrease $nml(T_i)$: to align the labels, we additionally need $nml(T_i)$ rename operations. Hence, the minimum number of edit operations that transform any instance of T_i to some instance of Q consists of (1) $nml(T_i)$ rename and (2) $|T_i| - |Q|$ delete operations, i.e., $\delta(Q, T_i) = nml(T_i) + |T_i| - |Q|$. Our assumption implies that $|T_i| - |Q| \geq \mathcal{B} - nml(T_i) + 1$. Therefore, $\delta(Q, T_i) \geq nml(T_i) + \mathcal{B} - nml(T_i) + 1 = \mathcal{B} + 1$, which contradicts our assumption.

Since the edit distance is symmetric, we do not need to consider the transformations of Q into T_i. □

For a given edit distance bound, $\mathcal{B}$, the subtrees within the size interval are called *pre-candidates*. The Cone algorithm proceeds in rounds. In every round some additional partitions are processed. Every round examines one additional list until all lists are initialized. We call a list *initialized* if we have already processed a partition in that list.

In the first round, $\mathcal{B} = 0$, and we process the partition of subtree size $|Q|$ in the first list (cf. Theorem 2.5.1). The subtrees in this partition can achieve an edit distance of 0 since their size matches the query size and all labels may match (no label mismatch found so far). Notably, these are the only subtrees that can achieve edit distance 0. Subtrees in other lists have at least one missing label w.r.t. Q, and subtrees in another partition of the first list are either smaller or larger than $|Q|$.

In every round $\mathcal{B}$ is incremented and an additional list is considered (if non-initialized lists are left). For the j-th list that we process, $nml(T_i) \geq j-1$: any new subtree T_i that we find in the j-th list has at least $j-1$ missing labels since we have processed all subtrees of size $|T_i|$ in the previous $j-1$ lists and did not see T_i.

We process only a subset of the partitions in a given list and round, namely the partitions that satisfy the size interval of the current round. Figure 2.6 illustrates this partition-based traversal.

The Cone algorithm distinguishes between pre-candidates and candidates. We use our index structure to generate pre-candidates. In the i-th round, $\mathcal{B} = i-1$, and we only need to consider the first i lists in the index. Similar to MergeAll, we maintain two pointers, l and r, for each list, initialized to the partition of the subtree size closest to $|Q|$. The pointers are used to generate pre-candidates from a partition. Some pre-candidates may be promoted to candidates. A pre-candidate T_i gets promoted whenever its label lower bound is equal to $\mathcal{B}$. Candidates are verified immediately, whereas the remaining pre-candidates are stored in the lower bound cache (lbc) for verification in a later round (cf. Section 2.4.2).

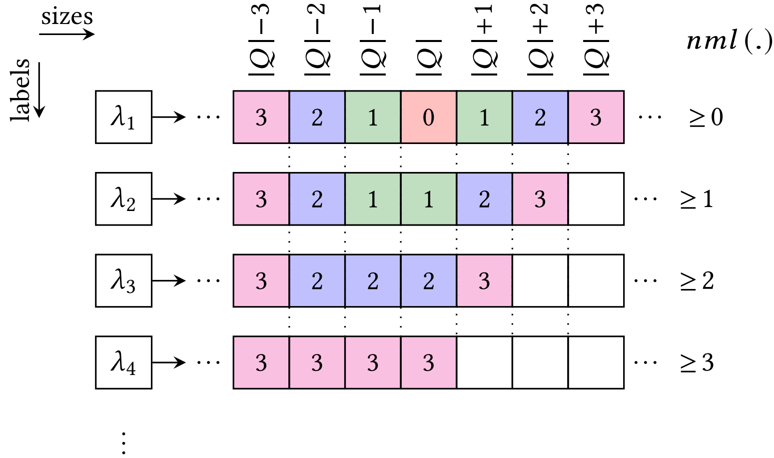


Figure 2.6: Cone traversal of the inverted list index in candidate score order.

INVERTED LIST ORDERING Since Cone examines lists one by one, the list order is important. Different pre-candidates may be reported for different list orders, resulting

in earlier/later termination as well as fewer/more label lower bound computations and verifications. Consider, for example, the lists in Figure 2.7 in reversed order $[l_b, l_x, l_a]$. Then, $llb(Q, T_{15}) = 3$ is computed in round 1 and T_{15} is cached for the round with $\mathcal{B} = 3$. Since the list length corresponds to the label frequency (a long list implies many subtrees with this label), we order the lists in ascending order by their length.

Like in MergeAll, we may not be able to produce enough candidates from the lists that share a label with the query. In this rare case, we fall back to MergeAll on all remaining lists to derive a correct ranking (cf. Section 2.4).

Theorem 2.5.2. *Cone solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Proof. Correctness: The stopping condition, $|R| = k \wedge \mathcal{B} \geq \delta(R[k])$, holds since all subtrees T_i in unprocessed partitions have size $|T_i| \notin si(\mathcal{B}, Q, T_i)$ and therefore $\delta(Q, T_i) > \mathcal{B}$ (cf. Theorem 2.5.1). Hence, these subtrees do not improve the ranking. If Cone does not produce enough candidates from lists that share a label with Q, we fall back to MergeAll on all remaining lists to derive a correct ranking. *Candidate score order*: We increment the edit bound $\mathcal{B}$, starting with $\mathcal{B} = 0$. For a given list x (starting with 0), we process all unprocessed partitions that contain subtrees in the size range $si(\mathcal{B}, Q, T_i)$. Let $P_{\mathcal{B}}$ denote the set of all subtrees T_i in these new partitions of the lists $x \leq \mathcal{B}$ that we have not seen before.

Similar to MergeAll, we perform two steps for a given $\mathcal{B}$: (1) We postpone the verification of subtrees $T_i \in P_{\mathcal{B}}$ for which $x = llb(Q, T_i) > \mathcal{B}$ and cache them in $lbc[x]$. (2) We verify (a) all subtrees $T_i \in P_{\mathcal{B}}$ for which $llb(Q, T_i) = \mathcal{B}$ and (b) all subtrees $T_i \in lbc[\mathcal{B}]$ (cached from previous sets $P_{\mathcal{B}'}$, $\mathcal{B}' < \mathcal{B}$). Hence, all subtrees T_i of the sets $P_{\mathcal{B}'}$, $\mathcal{B}' < \mathcal{B}$, with $llb(Q, T_i) = \mathcal{B}$ are verified when we process set $P_{\mathcal{B}}$. There is no subtree T_i with $llb(Q, T_i) \leq \mathcal{B}$ in some set $P_{\mathcal{B}''}$, $\mathcal{B}'' > \mathcal{B}$, i.e. $llb(Q, T_i) \leq \mathcal{B} \implies |T_i| \in si(\mathcal{B}, Q, T_i)$. Analogous to the proof of Theorem 2.5.1, we show this by contradiction. Recall that $llb(Q, T_i) = max\{|Q|, |T_i|\} - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$.

Case A: Assume a subtree T_i with $llb(Q, T_i) \leq \mathcal{B}$ and $|T_i| \leq |Q| - \mathcal{B} - 1$. Then, $max\{|Q|, |T_i|\} = |Q|$ implies that $llb(Q, T_i) = |Q| - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$. Our assumption yields $|\mathcal{L}(T_i)| \leq |Q| - \mathcal{B} - 1$, and $|\mathcal{L}(Q)| = |Q|$. Hence, $|\mathcal{L}(Q) \Cap \mathcal{L}(T_i)| \leq |Q| - \mathcal{B} - 1$ and therefore $llb(Q, T_i) \geq \mathcal{B} + 1$, which contradicts our assumption.

Case B: Assume a subtree T_i with $llb(Q, T_i) \leq \mathcal{B}$ and $|T_i| \geq |Q| + \mathcal{B} - nml(T_i) + 1$. Since $nml(T_i) \leq \mathcal{B}$, $max\{|Q|, |T_i|\} = |T_i| \implies llb(Q, T_i) = |T_i| - |\mathcal{L}(Q) \Cap \mathcal{L}(T_i)|$. Since $nml(T_i)$ labels of Q are not in T_i, $|\mathcal{L}(Q) \Cap \mathcal{L}(T_i)| = |Q| - nml(T_i)$ and therefore $llb(Q, T_i) = |T_i| - (|Q| - nml(T_i)) = |T_i| - |Q| + nml(T_i)$. Our assumption yields $|T_i| - |Q| \geq \mathcal{B} - nml(T_i) + 1$, hence $llb(Q, T_i) \geq \mathcal{B} + 1$, which contradicts our assumption.

In the fallback case, MergeAll guarantees score order. □

Example 2.5.3. *Figure 2.7 shows Cone applied on our running example, $k = 3$. The first round, $\mathcal{B} = 0$, retrieves and initializes l_a since l_a is the shortest list among all lists of the query labels. Pointer l is initialized to T_9, pointer r to T_6. Then, pre-candidates T_6, T_{11} are generated from partition 0 of l_a. Subtrees T_6 and T_{11} may match Q exactly since there is no label mismatch so far, and $|T_6| = |T_{11}| = |Q|$. Next, we compute the true label lower bounds using the node index; $llb(Q, T_6) = llb(Q, T_{11}) = 0$. Both are*

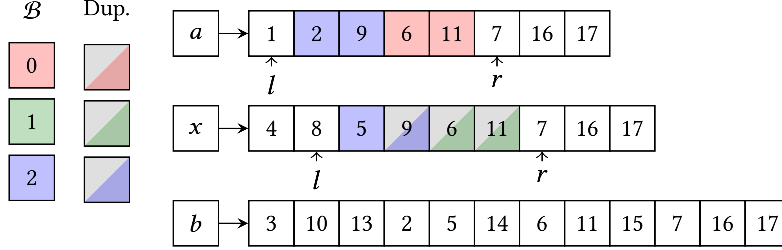


Figure 2.7: Processed subtrees of Cone.

verified and round 1 concludes; $R' = [T_6, T_{11}]$ *and* $\mathcal{B}$ *is incremented (lower bound cache lbc is empty). In round 2, we first process* l_a *again. The next partition of* l_a *contains subtrees of size* 2 *(*T_9, T_2*) and* 7 *(*T_7*), hence no pre-candidates are reported from* l_a*. Then, we initialize and process list* l_b *(l points to* T_9*, r to* T_6*), which does not provide us with new pre-candidates (*T_6, T_{11} *were already processed, indicated by the gray/green boxes). In round 3,* $\mathcal{B} = 2$*,* T_9 *and* T_2 *are reported from* l_a*, and* T_5 *is reported from* l_b*. All pre-candidates are promoted since* $llb\,(Q, T_9) = llb\,(Q, T_2) = llb\,(Q, T_5) = 2$*, resulting in* $R' = [T_6, T_5, T_9]$*. Since* $\mathcal{B} \geq \delta\,(R\,[k])$*, we terminate;* $R = [T_6, T_5, T_9]$*. Figure 2.7 depicts the processed list entries.*

Compared to MergeAll, Cone processes only 2 *lists (instead of* 4*) and computes only 4 label lower bounds. Notice how the presence of* T_{15} *in list* l_b *does not impose any overhead because we terminate before it is processed.*

PSEUDOCODE The pseudocode of Cone is given in Algorithm 4 (Algorithm 5 is an auxiliary function). It reuses Algorithms 2 and 3 from Section 2.4.2.

2.6 LINEAR SPACE INDEX AND SLIMCONE

Cone, presented in the previous section, is effective at producing candidates in score order. Unfortunately, Cone relies on an inverted list index that requires quadratic memory (in the worst case, cf. Section 2.4). In this section, we introduce the *slim inverted list* index, which requires only linear space, and the SlimCone algorithm that operates on the new index. SlimCone mimics Cone, but instead of scanning materialized inverted lists, relevant list parts are generated on the fly.

2.6.1 *Indexing in Linear Space*

In the worst case, the inverted list index requires quadratic space. To avoid the full materialization of the inverted lists, we introduce an implicit and lossless list representation that requires only linear space.

For a label $\lambda \in \mathcal{L}\,(T)$, the inverted list index stores every subtree that *contains* label λ. In other words, a list stores all nodes on every path from a node labeled λ up to the document root, and the paths are traversed at index build time. We propose *slim inverted lists* to avoid full list materialization and traverse paths during candidate

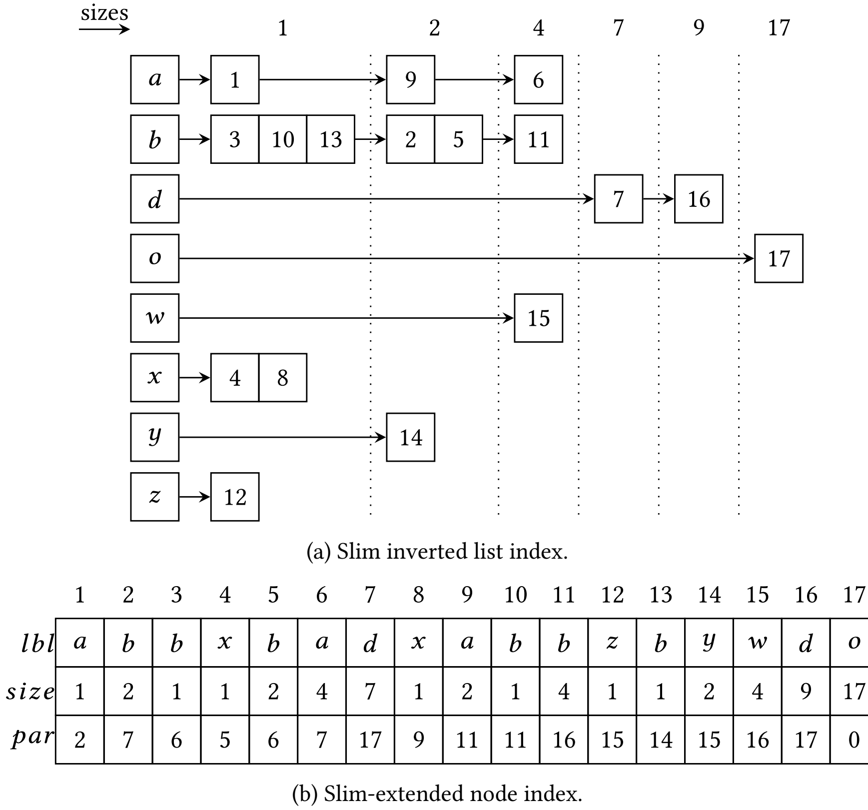
```
Algorithm 4: Cone(Q, T, k)
   Input: Query tree Q, document tree T, result size k
   Result: Top-k ranking R of subtrees of T w.r.t. Q
 1 L ← deduplicated ℒ(Q);
 2 sort L by increasing list length |l_λ|, λ ∈ L;
 3 lbc ← empty dynamic array; // lower bound cache
 4 ℬ ← 0; // current edit distance bound
 5 R ← empty ranking;
 6 while ℬ ≤ 2|Q| do // ℬ > 2|Q|: use MergeAll on all lists
 7     if Verify-Bucket(ℬ) then return R; // evaluate lbc[ℬ]
 8     foreach init. list l_λ do // process initialized lists first
 9         if Process-List(l_λ, ℬ) then return R;
10     if ℬ ≤ |L| then // initialize next list
11         l_λ ← I.list(L[ℬ]); // retrieve next list
           // process list l_λ; T_i ... subtree rooted at node i
12         if l_λ ≠ nil then // initialize pointers l_λ.r, l_λ.l
13             l_λ.r ← pos. of i s.t. ||Q| − |T_i|| is minimal;
14             l_λ.l ← l_λ.r − 1;
15             if Process-List(l_λ, ℬ) then return R;
16     ℬ ← ℬ + 1; // proceed to next ℬ' > ℬ
       // check if we can terminate before continuing
17     if |R| = k ∧ ℬ ≥ δ(R[k]) then return R;
18 return R;
```

```
Algorithm 5: Process-List(l_λ, ℬ)
   Input: Inverted list l_λ, edit distance bound ℬ
   Result: True if final ranking found, false otherwise
   // Q globally accessible; T_i ... subtree rooted at node i
 1 minsize ← |Q| − ℬ; // min. subtree size to consider
 2 maxsize ← |Q| + ℬ − idx[l_λ]; // max. subtree size to consider
 3 foreach unseen node i ∈ l_λ s.t. maxsize ≥ |T_i| ≥ minsize do
       // process T_i and return as soon as we can terminate
 4     if Process-Subtree(T_i, llb(T_i, Q), ℬ) then return true;
 5     advance l_λ.r and l_λ.l;
 6 return false; // we cannot terminate
```

generation. A slim inverted list (slim list) stores only nodes *labeled* λ (i.e., the start of a path). For the path traversals (upwards, towards the root node), we extend the node index (cf. Section 2.4.1) with parent pointers. This information enables us to reconstruct paths on the fly. Figure 2.8 depicts the slim inverted list index and the slim-extended node index of our running example.

2.6.2 *The SlimCone Algorithm*

We propose a new algorithm, SlimCone, that generates candidates in score order from slim inverted lists. Since we push the path traversals into the candidate generation



(a) Slim inverted list index.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lbl	*a*	*b*	*b*	*x*	*b*	*a*	*d*	*x*	*a*	*b*	*b*	*z*	*b*	*y*	*w*	*d*	*o*
size	1	2	1	1	2	4	7	1	2	1	4	1	1	2	4	9	17
par	2	7	6	5	6	7	17	9	11	11	16	15	14	15	16	17	0

(b) Slim-extended node index.

Figure 2.8: Linear-space index for example document T.

phase, SlimCone needs to walk up paths at query time using the slim-extended node index. SlimCone is also round-based ($\mathcal{B}$ is incremented in each round, starting with 0) and implements the Cone traversal on top of our slim inverted list index.

Cone can perform a binary search on the inverted lists to find the starting partitions. With slim lists, this approach would consider only nodes labeled λ, but there may be larger subtrees on the respective paths to the root. Slim lists do not store these subtrees explicitly. To generate correct pre-candidates, we need to traverse the respective paths for each entry of a slim list that represents a subtree smaller than Q. Notably, we may not need to traverse the paths completely, but only until we encounter a subtree T_i with size $|T_i| \geq |Q|$.

For the path traversal, we retrieve all node identifiers from the slim list at which a subtree T_i with $|T_i| < |Q|$ is rooted. For each identifier, we look up its parent in the node index and follow the path until the parent's subtree size is greater than or equal to $|Q|$. If the parent's subtree size is $|Q|$, then the parent is in the first partition; we immediately compute the label lower bound w.r.t. Q and verify the subtree if the label lower bound matches $\mathcal{B}$.

We keep track of the *path ends* (*pe*) for each slim list since we may need to continue the upward traversal in a later round. If the last node on the path roots a subtree that was verified, we store its parent in the path ends. Furthermore, we maintain a *path cache* (*pc*) for each slim list that stores all node identifiers on a path with the size of the

subtree they root. This avoids redundant traversals of the same path. Details on path cache and path ends are given below.

We may also need to examine additional list entries. Therefore, we store a single pointer for each list, *next*, which points to the next unprocessed list entry and is advanced whenever subtrees larger than Q are examined.

Note that the paths of *all* nodes that root a subtree T_i with $|T_i| < |Q|$ need to be traversed to generate all pre-candidates. While our algorithm climbs up all paths, it visits all nodes that root subtrees that are part of the corresponding full inverted list. Since we stop the traversal when we find a subtree root i s.t. $|T_i| \geq |Q|$, we construct the corresponding inverted list only partially. Figure 2.9 exemplifies this concept for example list l_b.

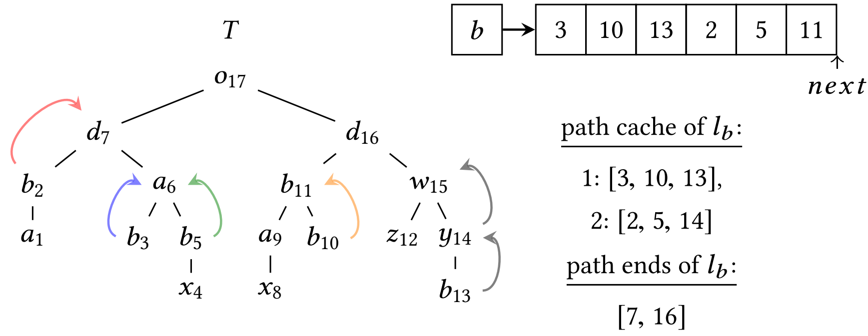


Figure 2.9: Finding the starting point of a slim list.

We discuss the main concepts used by SlimCone to generate candidates in non-increasing score order.

PATH CACHING The path cache (*pc*) stores a bucket for each subtree size that we encounter during the path traversals. In bucket b, we collect all roots of subtrees T_i s.t. $|T_i| = b$. This is necessary due to the vertical list expansion. Without the path cache, we would need to traverse the path downwards again. Hence, we reuse the path information in later rounds. If we need to consider smaller subtrees, we do a lookup in the path cache. This provides us with a (possibly empty) set of subtree roots, which contains all nodes that belong to a certain partition.

PATH ENDS We need to book-keep information about path ends (*pe*) for each slim list. After successfully climbing up a path to the first node at which a subtree T_i with $|T_i| \geq |Q|$ is rooted, we need to store the last node identifier on the path. This is due to the list expansion towards larger subtrees (w.r.t. $|Q|$). Therefore, for each slim list, we maintain a sequence of node identifiers, each of which represents the current end of a path. By storing these node identifiers, we can continue the upward path traversal in later rounds, if necessary.

LIST ORDERING To be consistent with the list ordering of Cone, we order the lists in SlimCone like in Cone, i.e., by increasing length. For each slim list, we compute and

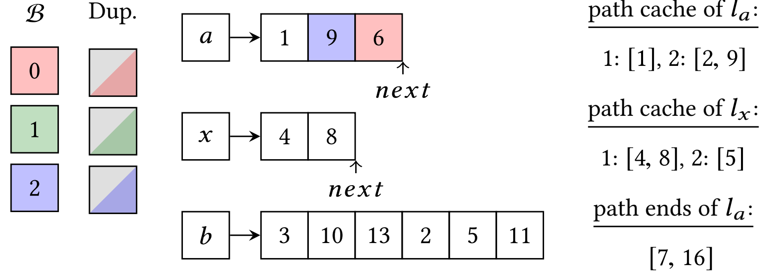


Figure 2.10: Slim lists, path caches, and path ends.

store the length of the corresponding full inverted lists. We refer to this value as *full list length.*

Similar to Cone, we use MergeAll on all lists of labels that are not in $\mathcal{L}(Q)$ to derive a correct ranking for the case that SlimCone produces too few results.

Theorem 2.6.1. *SlimCone solves the top-k subtree similarity problem and verifies subtrees in candidate score order.*

Proof. Correctness and *candidate score order* follow directly from Theorem 2.5.2 if we prove that SlimCone's partition traversal is identical to the partition traversal of Cone. Note that an identical partition traversal is sufficient, i.e., subtrees within the partitions need not be traversed in the same order. *Identical partition traversal*: SlimCone's list ordering is identical to the list ordering of Cone, hence lists (i.e., labels) are processed in the same order. We distinguish (1) uninitialized and (2) initialized lists: (1) Uninitialized lists: For each list entry i (rooting a subtree T_i) s.t. $|T_i| < |Q|$, the path in T is traversed upwards until $i' \neq i$ and $|T_{i'}| \geq |Q|$ holds. All traversed nodes (excl. i') are cached in the path cache pc. For $T_{i'}$, there are two cases: (a) $|T_{i'}| > |Q|$: i' is stored in the path ends pe. (b) $|T_{i'}| = |Q|$: $llb(Q, T_{i'})$ is computed. If $llb(Q, T_{i'}) = \mathcal{B}$, $T_{i'}$ is verified. Otherwise, we postpone the verification of $T_{i'}$ to round $\mathcal{B}'$. (2) Initialized lists: For a given list we process (a) all list entries i s.t. $|Q| + \mathcal{B} - nml(T_i) \geq |T_i| > |Q|$, (b) all entries in the path cache pc, and (c) all entries in the path ends pe. Due to (b) we process all subtrees T_i smaller than Q, $|Q| > T_i \geq |Q| - \mathcal{B}$; due to (c) we process all subtrees T_i larger than Q, $|Q| + \mathcal{B} - nml(T_i) \geq T_i > |Q|$. (2) and (3) guarantee that (i) SlimCone's partition traversal is identical to the partition traversal of Cone *and* (ii) *all* subtrees of a partition are generated. □

Example 2.6.2. *In Figure 2.10, we illustrate SlimCone for our running example, $k = 3$. Similar to Cone, we retrieve slim list l_a since it is the shortest w.r.t. the full list lengths. The initialization for l_a now differs from Cone: we climb up the paths of all entries of l_a since the subtree sizes are smaller than or equal to $|Q|$. This results in the path cache and path ends of l_a shown in Figure 2.10. During the traversal, we find T_{11} and T_6 (in this order) having $|T_6| = |T_{11}| = |Q|$. Consequently, we compute $llb(Q, T_6) = llb(Q, T_{11}) = 0$ and verify both, $\delta(Q, T_6) = 0$, $\delta(Q, T_{11}) = 4$. This results in $R' = [T_6, T_{11}]$. Note that after examining T_6 and T_{11}, we traverse to their respective parents. Therefore, T_{16} is added to the path ends of l_a. No new pre-candidates are processed in round 2. However, list l_x is retrieved*

and initialized, resulting in the path cache and path ends of l_x depicted in Figure 2.10. Since we have already stored the node identifiers 9, 7, and 16 during initialization of l_a, neither 9 is added to the path cache nor 7, 16 are added to the path ends of l_x. In round 3, $\mathcal{B} = 2$, we process bucket 2 of the path cache of l_a, generating the pre-candidates T_2 and T_9. We compute $llb(Q, T_2) = llb(Q, T_9) = 2$, verify T_2 and T_9, and update $R' = [T_6, T_9, T_2]$. Analogously, we process T_5 from the path cache of l_x. This results in $R = [T_6, T_5, T_9]$.

PSEUDOCODE Algorithm 6 provides the pseudocode of SlimCone (Algorithms 7 and 8 are auxiliary functions). Again, Algorithms 2 and 3 are reused (cf. Section 2.4.2).

2.7 EFFICIENT INDEX UPDATES

We extend the slim index to support incremental updates. We support the standard node edit operations as defined in Section 2.2: rename, delete, and insert. Updates affect the inverted list index and the node index.

UPDATING THE INVERTED LIST INDEX The position of a node in the inverted list index is determined by its label and the size of its subtree. The rename operation changes the label of a node, which requires us to remove the node from the list of the old label and insert it into the list of the new label. Insert (delete) changes the subtree size of all ancestors of the inserted (deleted) node, and we must add the new node into the respective list (remove the deleted node). We implement a slim inverted list as a balanced search tree (ordered by subtree size), thus requiring only $O(\log l)$ time to find, insert, or delete a node from a list of length l. No further changes are required to the slim inverted list index in response to node edit operations on the document. Space and runtime complexity at query time are not affected.

DYNAMIC NODE INDEX The node index encodes the labels and the structure of the document T. At query time, we need to efficiently support the following operations: (1) access a node by its identifier, (2) reconstruct a subtree (or the label set of a subtree) given its root node. A subtree is reconstructed by traversing all its nodes in postorder (cf. Section 2.4.1).

(1) In the static node index, we identify a node by its postorder position. In our dynamic version of the slim-extended node index, we allow arbitrary node identifiers. The node index is stored in an array and the identifier of a node matches the array position, thus a node is accessed in constant time. To ensure a compact representation, we use consecutive identifiers and maintain a free list to reuse array positions after node deletions.

(2) To reconstruct a subtree given its root node, we store two additional fields for each node v: first child, $c1(v)$; next (right) sibling, $sib(v)$. These fields also allow us to efficiently traverse all nodes of a subtree in postorder.

We discuss the effect of updates on the dynamic node index. *Rename:* The node is accessed via its identifier and the label is changed in constant time. *Insert:* The insert operation adds a new node u as the i-th child of an existing node p, replacing a

Algorithm 6: SlimCone(Q, T, k)

```
   Input: Query tree Q, document tree T, result size k
   Result: Top-k ranking R of subtrees of T w.r.t. Q
 1 L ← deduplicated 𝓛(Q);
 2 sort L by increasing full list length |l_λ|, λ ∈ L;
   // lower bound cache lbc, path cache pc, path ends pe, and
   // positions in slim lists next
 3 lbc, pc, pe, next ← empty dynamic arrays;
 4 𝓑 ← 0; // current edit distance bound
 5 R ← empty ranking;
 6 while 𝓑 ≤ 2|Q| do // 𝓑 > 2|Q|: use MergeAll on all lists
 7     if Verify-Bucket(𝓑) then return R; // evaluate lbc[𝓑]
 8     foreach init. list l_λ do // process initialized lists first
 9         if Process-Smaller(l_λ, 𝓑) then return R;
10         if Process-Larger(l_λ, 𝓑) then return R;
11     if 𝓑 ≤ |L| then // initialize next list
12         l_λ ← 𝓘.list(L[𝓑]); // retrieve next list
13         if l_λ ≠ nil then
               // initialize l_λ buckets in pc and pe
14             pc[l_λ], pe[l_λ] ← empty dynamic Arrays;
15             next[l_λ] ← 0; // initialize next pointer for l_λ
16             foreach unseen node i ∈ l_λ s.t. |T_i| < |Q| do
                   // climb up path; T_q ... subtree rooted at q
17                 traverse up to first node q s.t. |T_q| ≥ |Q|;
                   // add all traversed nodes to path cache
18                 foreach traversed node x (excl. q) do
19                     pc[l_λ][|T_x|] ← pc[l_λ][|T_x|] ∪ {x};
                   // process T_q if size fits |Q|
20                 if q unseen ∧ |T_q| = |Q| then
21                     lb ← llb(T_q, Q);
22                     if Process-Subtree(T_q, lb, 𝓑) then return R;
23                     q ← par(q); // traverse to parent of q
24                 pe[l_λ] ← pe[l_λ] ∪ {q}; // add q to path ends
25                 next[l_λ] ← pos. of i in l_λ; // next l_λ-entry
26             if Process-Smaller(l_λ, 𝓑) then return R;
27             if Process-Larger(l_λ, 𝓑) then return R;
28     𝓑 ← 𝓑 + 1; // proceed to next 𝓑' > 𝓑
       // check if we can terminate before continuing
29     if |R| = k ∧ 𝓑 ≥ δ(R[k]) then return R;
30 return R
```

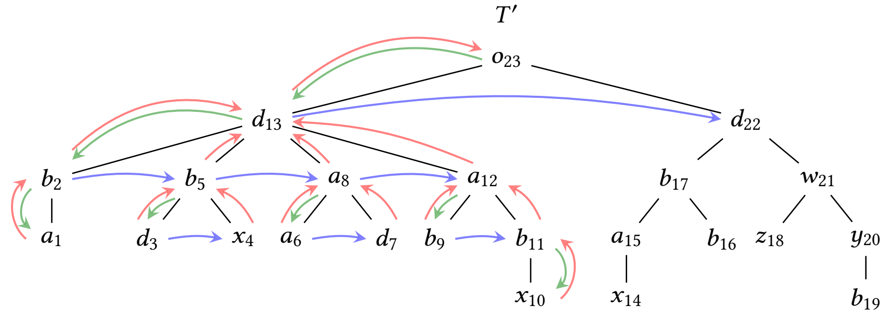
(possibly empty) sequence $C = (c_i, c_{i+1}, \ldots, c_j)$ of $p's$ children, and the child sequence is connected under the new node u. We need to insert a new node into the index; the identifier of the new node matches its position in the node index array (new nodes are appended or fill a gap resulting from an earlier deletion). The following existing nodes must be updated. (a) Ancestors of the inserted node u: The subtree sizes are incremented by one; if u is the first child of its parent p, the first child pointer, $c1(p)$, is updated. (b) (Former) children of p: The parent pointer of all nodes in C is updated; the

```
Algorithm 7: Process-Smaller(l_λ, B)
  Input: Inverted list l_λ, edit distance bound B
  Result: True if final ranking found, false otherwise
  // pc, Q globally accessible
1 minsize ← |Q| − B; // min. subtree size to consider
  // process all path cache buckets that fit w.r.t. size
2 s ← |Q| − 1;
3 while s ≥ minsize do
4     b ← pc[l_λ][s]; // get path cache bucket
      // process all subtrees; T_q ... subtree rooted at node q
5     foreach unseen node q ∈ b do
          // process T_q and return as soon as we can terminate
6         if Process-Subtree(T_q, llb(T_q, Q), B) then
7             return true;
8     s ← s − 1; // proceed to next subtree size
9 return false; // we cannot terminate
```

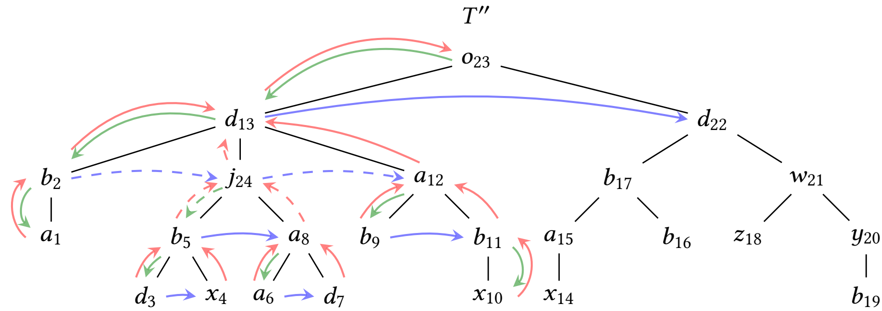
```
Algorithm 8: Process-Larger(l_λ, B)
  Input: Inverted list l_λ, edit distance bound B
  Result: True if final ranking found, false otherwise
  // next, pe, Q globally accessible
1 maxsize ← |Q| + B − idx[l_λ]; // max. subtree size to consider
  // add fitting list entries to path ends bucket
2 while T_n ← subtree rooted at next[l_λ] ∧ |T_n| ≤ maxsize do
3     pe[l_λ] ← pe[l_λ] ∪ {next[l_λ]};
4     advance next[l_λ];
5 foreach unseen node q ∈ pe[l_λ] do // process path ends bucket
6     if |T_q| ≤ maxsize then // T_q ... subtree rooted at node q
          // process T_q and return as soon as we can terminate
7         if Process-Subtree(T_q, llb(T_q, Q), B) then
8             return true;
9         q ← par(q); // traverse to parent
10 return false; // we cannot terminate
```

next sibling pointers of c_j and (if $i > 1$) c_{i-1} are updated. To insert a new root node, we assume a virtual node with identifier zero, which is treated as the parent of the actual root node. *Delete* is the reverse of insert. The positions of deleted nodes are registered in the free list.

Figure 2.11 illustrates the slim inverted lists and the dynamic node index after inserting a new node into an example tree T'. Only the dashed pointers in T'' (colored fields in slim lists and dynamic node index) need to be updated. Red, green, and blue pointers (fields) denote the parent, first child, and next sibling pointers (fields), respectively. Changes to subtree sizes are highlighted in gray. Overall, the complexity of updating the node index is $O(d + f)$, where d is the depth and f the maximum



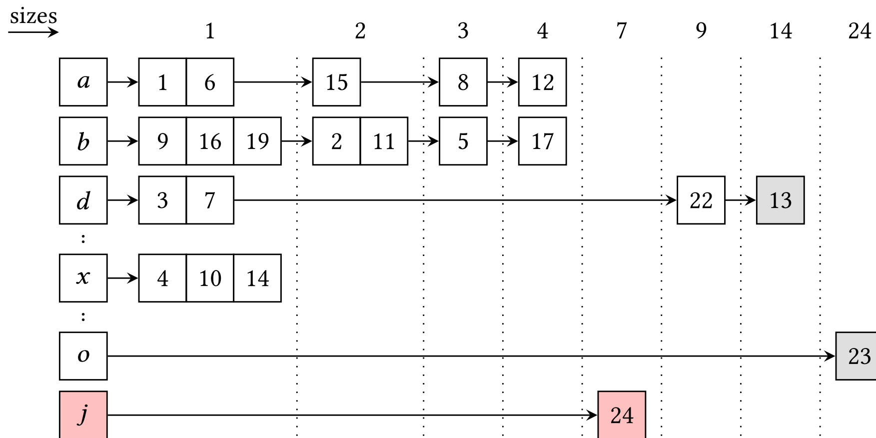
(a) Example tree before node insertion (T').



(b) Example tree after inserting node j_{24} (T'').

	1	2	3	4	5	6	7	8	9	10	11	12	13		23	24
lbl	a	b	d	x	c	u	p	v	b	x	b	a	d		o	j
$size$	1	2	1	1	3	1	1	3	1	1	2	4	14		24	7
par	2	13	5	5	24	8	8	24	12	11	12	13	23	..	0	13
$c1$	0	1	0	0	3	0	0	6	0	0	10	9	2		13	5
sib	0	24	4	0	8	7	0	0	11	0	0	0	22		0	12

(c) Dynamic node index of example tree T''.



(d) Slim inverted list index of example tree T''.

Figure 2.11: Index update example.

fanout of a node in the document. In many real datasets, f and d are small compared to the document size. We show the efficiency of updates in our experiments.

2.8 RELATED WORK

Top-k Subtree Similarity Queries. TASM-Dynamic [8, 136], a simple solution for the top-k subtree similarity problem, computes the edit distance between the query Q and the entire document T using dynamic programming. As a side product, the edit distances between the query and all subtrees of the document are computed. This approach requires $O(|Q|^2 |T|)$ time and $O(|Q| |T|)$ space [35]. Augsten et al. [8, 9] show that the maximum subtree size that must be considered is $\tau = 2|Q| + k$. They develop the TASM-Postorder algorithm that runs in $O(|Q|^2 + |Q| k)$ space, i.e., the memory is independent of the document size. TASM-Postorder does not use an index and must scan the document for each query. We empirically compare our solution to TASM-Postorder.

Cohen [31] proposes StructureSearch, the first index-based method for top-k subtree similarity queries. The index identifies repeating subtree patterns to reduce the number of redundant edit distance computations. StructureSearch does not need to scan the document at query time and outperforms TASM-Postorder in terms of runtime. However, StructureSearch requires a large index, which can be quadratic in the document size. The document is the database, which may be large (e.g., SwissProt has $|T| = 479$M nodes). Our SlimCone algorithm requires only a linear-size index. We empirically compare StructureSearch to SlimCone. Our solution builds a smaller index, building the index is faster, and in most settings we outperform StructureSearch in terms of query response time, often by orders of magnitude.

XML Indexing Techniques. Inverted lists and data structures similar to our node index have also been used to index XML documents [51, 66]. These works solve a different problem (answering resp. ranking XPath queries) and do not consider the tree edit distance. Further, our index access methods are different: we access the inverted lists partition by partition based on an edit distance bound and build the partitions on the fly while accessing them.

Tree Edit Distance. The classical tree edit distance algorithm by Zhang and Shasha [136] runs in $O\left(n^4\right)$ time and $O\left(n^2\right)$ space for trees with n nodes; for flat trees of depth $O(\log n)$ the algorithm runs efficiently in $O\left(n^2 \log^2 n\right)$ time. Bille [18] surveys classical edit distance algorithms. Newer developments include the algorithm by Demaine et al. [35], which reduces the runtime to $O\left(n^3\right)$, and AP-TED$^+$ by Pawlik and Augsten [91]. AP-TED$^+$ analyzes the input trees and dynamically computes the optimal evaluation strategy. While the runtime complexity remains cubic, this worst case can often be avoided. Despite all efforts, computing the edit distance remains expensive. We introduce the candidate score to rank subtrees, verify promising candidates first, and thus reduce the number of expensive edit distance computations.

Related Problem Definitions. Related but different problems include, for example, XML duplicate detection [26, 95], approximations of the tree edit distance [12, 133],

tree similarity joins [111], top-k similarity joins for sets [129], and top-k queries over relational data [60].

Cohen et al. [30] introduce a top-k algorithm that works for both ordered and unordered trees. In near linear time, the algorithm retrieves the top-k subtrees in a document w.r.t. a so-called profile distance function. A profile distance function projects tree features to a multiset and evaluates the distance between feature multisets; examples include pq-grams [12] and binary branches [133]. The algorithm by Cohen et al. [30] solves a related but different problem and does not provide edit distance guarantees on the ranking.

In their TA algorithm, Fagin et al. [42, 43] process ranked lists of items sorted by some local score. The global score of an item is computed based on the respective local scores. The goal is to find the k items with the highest global score. Akbarinia et al. [2] improve the efficiency of TA by minimizing the number of list accesses. Theobald et al. propose TA-based solutions to answer probabilistic top-k queries [115], efficiently expand queries [113], and build an efficient top-k query processing system for semi-structured data [116]. We introduce the candidate score on subtrees, but we do not merge ranked lists. The challenge in our setting is to rank candidates efficiently and produce the head of the ranked list without generating the tail.

2.9 EMPIRICAL EVALUATION

We empirically compare our solutions to two state-of-the-art algorithms on both synthetic and real-world data. We vary document size, query size, and k, and measure query time, indexing time, main memory, and the number of verifications.

2.9.1 *Setup & Datasets*

SETUP All experiments were conducted on a 64-bit machine with 8 Intel(R) Xeon(R) CPUs E5-2630 v3, 2.40GHz, 20MB L3 cache (shared), 256KB L2 cache (per core), and 96GB of RAM, running Debian 8.11, kernel 3.16.0-6-amd64. We compile our code with `clang` (ver. 3.5.0-10) at maximum optimization level (`-O3`). Although we have multiple cores, we run all experiments single-threaded with no other load on the machine. We measure the runtime with `getrusage`[2] (sum of user and system CPU time). Each runtime measurement is an average over five runs. We measure main memory as the heap peak value provided by the `libmemusage.so` library[3] (preloaded using the `LD_PRELOAD` environment variable).

DATASETS AND QUERIES We use the XMark benchmark to generate synthetic datasets of five different sizes. Additionally, we run experiments on three real-world

2 http://man7.org/linux/man-pages/man2/getrusage.2.html
3 http://man7.org/linux/man-pages/man1/memusage.1.html

datasets: TreeBank[4] (TB), DBLP[5], and SwissProt[6] (SP). Important dataset characteristics are summarized in Table 2.3. XMark, DBLP, and SwissProt were also used in previous work [31], although only small subsets of DBLP and SwissProt were used; we process the full datasets. From each of the datasets (documents, T), we randomly extract four different queries, Q, with 4, 8, 16, 32, 64 nodes, respectively. We also vary the result size, k.

Table 2.3: Dataset characteristics.

Name	Size T [MB]	Size [Nodes] $\lvert T\rvert$	Size [Nodes] avg. $\lvert T_i\rvert$	# diff. labels
XMark1	112	$3.6 \cdot 10^6$	6.2	$510 \cdot 10^3$
XMark2	223	$7.2 \cdot 10^6$	6.2	$822 \cdot 10^3$
XMark4	447	$14.4 \cdot 10^6$	6.2	$1.3 \cdot 10^6$
XMark8	895	$28.9 \cdot 10^6$	6.2	$1.9 \cdot 10^6$
XMark16	1,790	$57.8 \cdot 10^6$	6.2	$2.9 \cdot 10^6$
TreeBank	83	$3.8 \cdot 10^6$	8.4	$1.4 \cdot 10^6$
DBLP	2,161	$126.5 \cdot 10^6$	3.4	$21.6 \cdot 10^6$
SwissProt	6,137	$479.3 \cdot 10^6$	5.1	$11.4 \cdot 10^6$

ALGORITHMS We compare our algorithms MERGE, CONE, SLIM (cf. Sections 2.4–2.6) to the state-of-the-art algorithms TASMPostorder [8, 9] (TASM, fastest index-free algorithm) and StructureSearch [31] (STRUCT, fastest algorithm with precomputed index). SLIM-DYN refers to the version of SLIM with incremental update support (cf. Section 2.7). All algorithms were implemented in C++11. We maintain the node labels in a dictionary and replace string labels by integers. All indexes reside in main memory. For computing the tree edit distance, we use the algorithm by Zhang & Shasha [136], which is efficient for flat trees (depth $O(\log n)$, as is typically the case in XML).

SPACE-EFFICIENT STRUCTURESEARCH Cohen [31] implements the StructureSearch algorithm using uncompressed Dewey labels (STRUCT-DEWEY in our experiments), which leads to large indexes of about 10 times the document size (in MB); in the worst case, the index size is quadratic in the document size since each Dewey label may be of linear size. Cohen suggests to compress the Dewey labels to improve space performance. We take a different approach and use preorder, postorder, and parent (preorder) identifiers to (1) verify ancestor relationships and (2) to generate the ancestor path of a node. Given the pre- and postorder identifiers of two nodes u and v, u is an ancestor of v if and only if $pre(v) > pre(u) \wedge post(v) < post(u)$. We efficiently generate the path between a node v and its ancestor u using the parent pointers. Thus, the node identifiers in our space-efficient implementation (STRUCT) have constant size and we need not deal with compressed Dewey labels to verify node relationships or

4 https://www.seas.upenn.edu/~pdtb/
5 https://dblp.uni-trier.de/xml/dblp.xml.gz
6 ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.xml.gz

generate ancestor paths. In our experiments, we show that we substantially reduce the index size w.r.t. the original implementation.

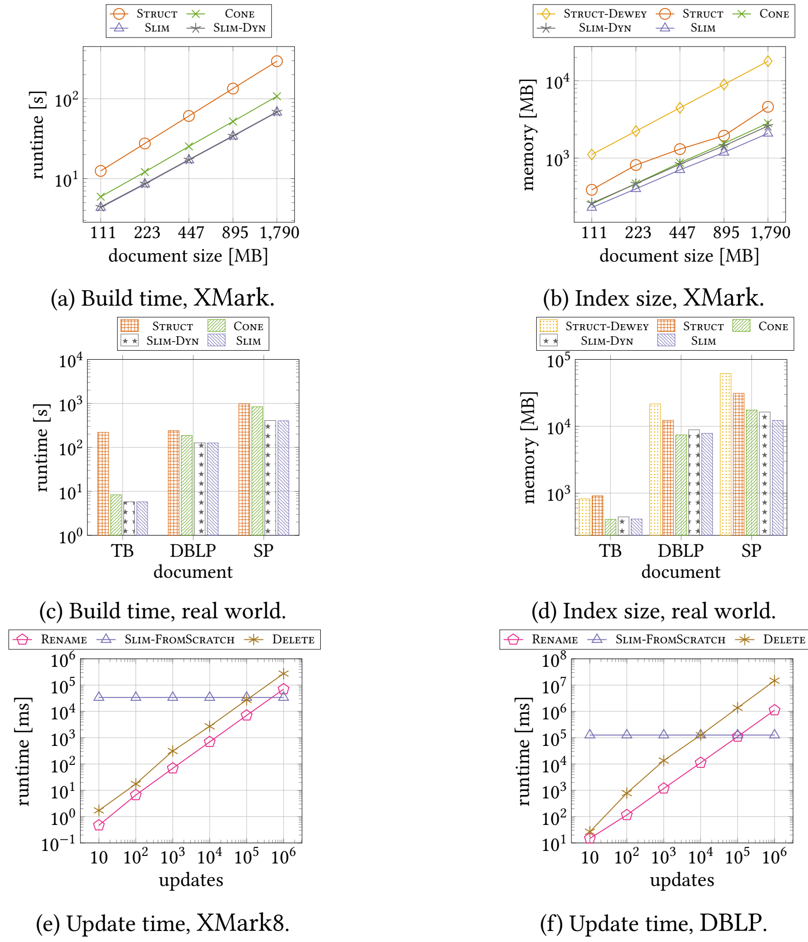
Struct assumes XML and distinguishes common and uncommon labels. Inner nodes and the x most frequent leaf nodes of an XML document are considered common. Further, Struct has a maximum edit bound y. The top-k ranking of Struct contains only subtrees with a maximum edit distance of y. All datasets in our tests are available in XML, thus we configure Struct as suggested by Cohen [31] and set $x = 1000$, $y = |Q|$. Our algorithms do not require any parameters.

2.9.2 *Indexing*

We compare the indexes of Struct, Cone, Slim, and Slim-Dyn in terms of size (all memory-resident index structures including the document) and runtime to build the index. With Struct-Dewey we refer to the original implementation of Struct by Cohen [31], which uses uncompressed Dewey labels. The index of Merge is identical to the index of Cone and is not shown separately; Tasm does not build an index.

The results are shown in Figure 2.12. For Struct-Dewey, we estimate the index size based on the instructions of Cohen [31] (index size is about 10 times the document size). Our space-efficient implementation of Cohen's algorithm (Struct) substantially improves the memory and requires only about 2–5 times the document size (except for TB). Cone and Slim clearly outperform Struct both in terms of index size and runtime for building the index. Even the space-efficient implementation of Struct requires at least 1.5–3 times more memory than Slim. Except for DBLP and TB, the index size of Slim is within two times the document size. Among our algorithms, Slim is faster and builds a smaller index. This is expected since Slim indexes each node once, while Cone may index each node multiple times. In the worst case, when the depth of the document grows linearly with its size, the index of Cone grows quadratically; this is not the case for the documents in our test. The size of Slim-Dyn (which supports incremental updates) is similar to the size of the space-efficient implementation of Struct (which does not support updates), but builds much faster.

INCREMENTAL UPDATES We compare the time to incrementally update the slim index to the time of building the static slim index from scratch (Slim-From-Scratch). Figure 2.12e and Figure 2.12f show the results for the XMark8 and the DBLP datasets, respectively. We randomly rename or delete nodes in the document. Insertion is similar to deletion in that it reverses the index updates of a delete operation. The update time is linear in the number of updates for both rename and deletion. As expected, deletion takes slightly more time than rename since all ancestors and children of the deleted node must be updated. The break even point for building the index from scratch is at about 10^5 deletions / $5 \cdot 10^5$ renames for XMark8 and 10^4 deletions / 10^5 renames for DBLP.



(a) Build time, XMark.

(b) Index size, XMark.

(c) Build time, real world.

(d) Index size, real world.

(e) Update time, XMark8.

(f) Update time, DBLP.

Figure 2.12: Build time, index size, and update time.

2.9.3 *Effectiveness and Query Time*

We evaluate our algorithms that process the subtrees in candidate score order (Merge, Cone, Slim, Slim-Dyn, cf. Figure 2.13). Since Slim and Slim-Dyn are the same algorithms operating on different indexes, we only discuss Slim. Supporting updates only marginally affects the query time for varying query, document, and result size (cf. Slim-Dyn in Figures 2.13–2.16). Merge needs to verify many more candidates and is consistently slower than its competitors. This confirms the effectiveness of the clever list traversal used by Cone and Slim. In some cases, Cone is faster than Slim since Slim must build the lists on the fly; we measure the largest difference for DBLP, where Slim must traverse many paths to initialize the inverted lists. The number of verifications is the same for both algorithm. Overall, the runtime difference is small in most cases, thus Slim pays a low price for reducing the memory complexity from quadratic to linear.

Next, we compare Slim to the two state-of-the-art approaches with precomputed index (Struct) resp. without index (Tasm). The query time increases with the document size for all solutions except Slim (cf. Figure 2.14). The runtime of Slim may even decrease with the document size. Larger documents have more subtrees, therefore

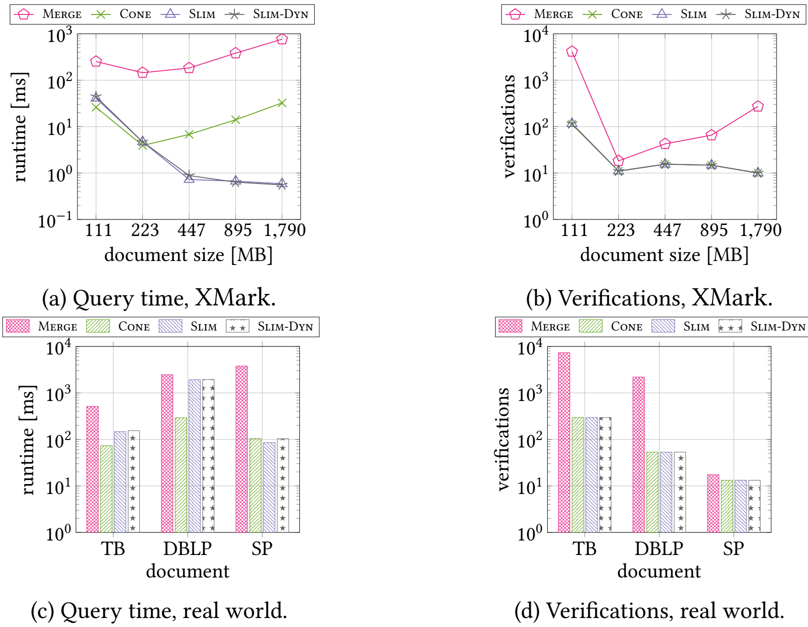


Figure 2.13: Merge, Cone, Slim: Query time and number of verifications over document size, $k = 10$, $|Q| = 16$.

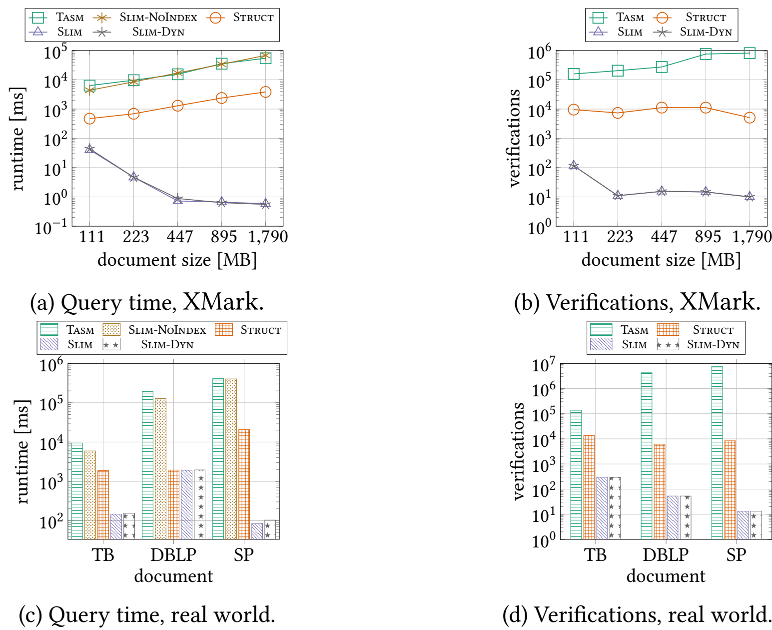
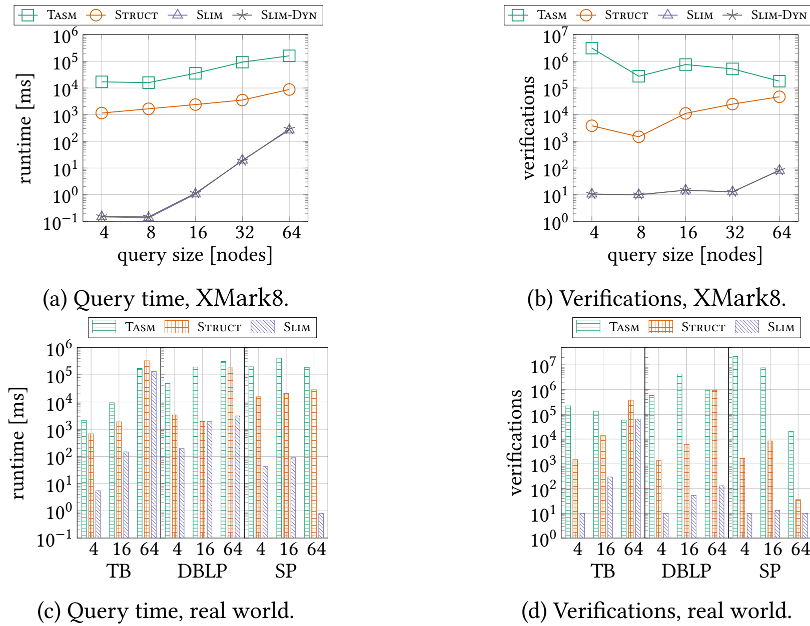


Figure 2.14: State of the art vs. Slim: Query time and number of verifications over document size, $k = 10$, $|Q| = 16$.

there is a better chance to fill the ranking with good matches and terminate early. For example, the number of verifications decreases between XMark1 and XMark2. Slim builds and traverses only the relevant parts of the lists and is therefore efficient for large documents. Also for Struct, the number of verifications is independent of the

document size, but in absolut numbers Slim verifies between two and three orders of magnitude fewer candidates. Further, the runtime of Struct substantially increases with the document size. Overall, Slim is up to three orders of magnitude faster than Struct. Notably, Slim is beneficial for a single query even without precomputed index (cf. Slim-NoIndex in Figures 2.14a and 2.14c).

Slim outperforms its competitors also when the query size increases (Figure 2.15). Note the small number of verifications in Figure 2.15b: for $|Q| = 8$, Slim verifies only k candidates, which is optimal (only subtrees that appear in the final ranking are verified). This confirms the effectiveness of the score order and the clever list traversal in Slim. Struct resp. Tasm must verify at least two resp. three orders of magnitude more candidates (except for TB, $|Q| = 64$). The runtime of Slim on XMark8 is always below $1s$ ($|Q| = 4$ and $|Q| = 8$: below $1ms$), whereas the best competitor, Struct, runs for at least $1s$ and up to $8s$. The results on our real-world datasets lead to similar conclusions; only on DBLP Struct is slightly faster.



(a) Query time, XMark8.

(b) Verifications, XMark8.

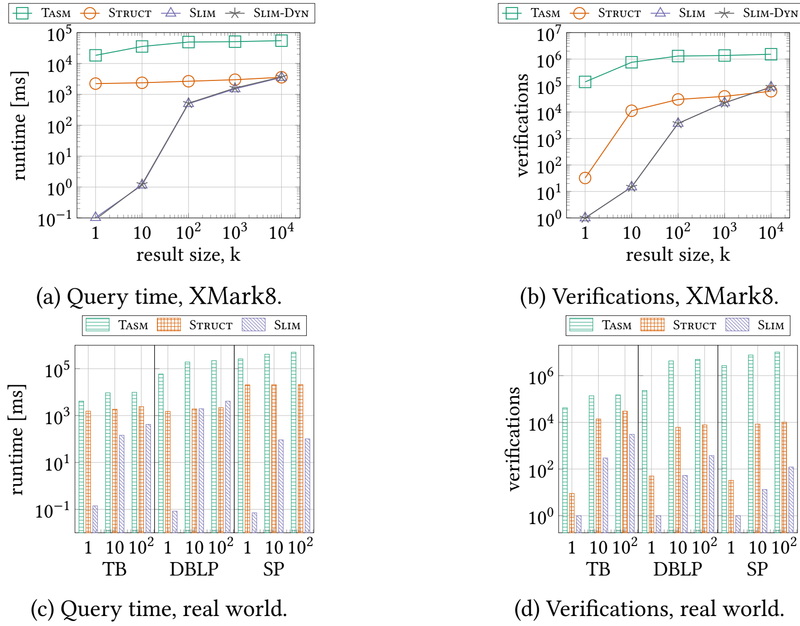
(c) Query time, real world.

(d) Verifications, real world.

Figure 2.15: State of the art vs. Slim: Query time and number of verifications over query size $|Q|$, $k = 10$.

In Figure 2.16, we vary the result size k. All algorithms produce more candidates since the lower bound computed from the top-k ranking is looser when k is larger (and thus the subtree at position k in the ranking is less similar to the query). Slim benefits from the small candidate set for small values of k and achieves runtimes between $0.1ms$ and $1s$ in the range $k = 1$ to $k = 100$. Struct must verify many more candidates than SlimCone. Although in Struct the number of verifications for $k = 1$ is by orders of magnitude smaller than for $k = 100$, the runtime improves only marginally. Struct retrieves many subtrees from the index that are filtered before they are verified; the number of retrieved subtrees does not depend on k and may be much larger than the number of verifications. Slim does not incur this overhead: candidates are processed

partition by partition, and more promising partitions are processed first. Except for DBLP, Slim outperforms Struct on all k values except $k = 10000$. For $k = 10000$, both algorithms must verify many subtrees. Struct groups subtrees into equivalence classes of subtrees and verifies only one representative in each class, thus saving edit distance computations. This verification technique is orthogonal to the candidate generation and could also be adopted in Slim.



(a) Query time, XMark8. (b) Verifications, XMark8.

(c) Query time, real world. (d) Verifications, real world.

Figure 2.16: State of the art vs. Slim: Query time and number of verifications over varying result size k, $|Q| = 16$.

2.10 CONCLUSION

In this chapter, we introduced a novel indexing technique for top-k subtree similarity queries, which retrieve the k most similar subtrees in a document T w.r.t. a query tree Q. We proposed the first incrementally updatable, linear-space index to solve this problem. Previous solutions either scan the entire document, or they build an index, but the index is static (not updatable) and large (quadratic in the document size in the worst case). The document is the database on which we compute the top-k query. Computing a quadratic-size index is not feasible for large documents.

We proposed the candidate score, which sorts subtrees such that more promising subtrees appear earlier in the sort order. We could show that processing subtrees in candidate score order substantially reduces the number of items that must be processed and verified. We developed SlimCone, a novel algorithm that leverages our linear-size index to efficiently retrieve candidates in non-increasing candidate score order. SlimCone is not tailored to XML (like previous work) and does not require any tuning parameters.

Our experiments confirmed the effectiveness of our techniques. SlimCone outperformed the state of the art in almost all scenarios w.r.t. memory consumption, number of verifications, indexing time, and query time, often by multiple orders of magnitude.

ACKNOWLEDGMENTS We thank the anonymous reviewers for their constructive comments and Mateusz Pawlik, Thomas Hütter, and Willi Mann for valuable discussions. This work was supported by the Austrian Science Fund (FWF): P 29859.

3 SCALING DENSITY-BASED CLUSTERING TO LARGE COLLECTIONS OF SETS

AUTHORS Daniel Kocher, Nikolaus Augsten, and Willi Mann
VENUE EDBT, Nicosia, 2021 (Accepted)[1]

ABSTRACT

We study techniques for clustering large collections of sets into DBSCAN clusters. Sets are often used as a representation of complex objects to assess their similarity. The similarity of two objects is then computed based on the overlap of their set representations, for example, using Hamming distance. Clustering large collections of sets is challenging. A baseline that executes the standard DBSCAN algorithm suffers from poor performance due to the unfavorable neighborhood-by-neighborhood order in which the sets are retrieved. The DBSCAN order requires the use of a symmetric index, which is less effective than its asymmetric counterpart. Precomputing and materializing the neighborhoods to gain control over the retrieval order often turns out to be infeasible due to excessive memory requirements.

We propose a new, density-based clustering algorithm that processes data points in any user-defined order and does not need to materialize neighborhoods. Instead, so-called backlinks are introduced that are sufficient to achieve a correct clustering. Backlinks require only linear space while there can be a quadratic number of neighbors. To the best of our knowledge, this is the first DBSCAN-compliant algorithm that can leverage asymmetric indexes in linear space. Our empirical evaluation suggests that our algorithm combines the best of two worlds: it achieves the runtime performance of materialization-based approaches while retaining the memory efficiency of non-materializing techniques.

3.1 INTRODUCTION

We consider the problem of partitioning large collections of sets into DBSCAN clusters [40]. Our work is motivated by a process mining use case at Celonis SE that models processes as sets. A *process* is a sequence of timestamped activities. Large companies store hundreds of millions of activities in millions of processes. In order to analyze the processes, they should be clustered into groups of similar activity sequences that can be further explored [22, 58, 105]. To this end, a process is represented by the set of all its neighboring activity pairs, e.g., the process with the activity sequence (S, O, P, H, R, F, E) (Start, Order, Pay, sHip, Return good, reFund, End) is represented

1 The original publication Kocher et al. [70] (EDBT 2021) discusses a multi-core extension. This extension can be found in Section 4.2 of this thesis.

by the set $\{(S,O),(O,P),(P,H),(H,R),(R,F),(F,E)\}$. The similarity of two processes is then assessed by the Hamming distance[2] of their set representations.

Sets are used in many other applications [81] to represent objects for the purpose of clustering, e.g., sales may be represented by sets of product categories, photos by sets of tags and title words, user interactions on a website by sets of visited links, users of a social network by their group memberships, or users of a music streaming platform by sets of tracks they listen to.

The popular DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm [40] identifies clusters of arbitrary shape without requiring the number of clusters as input. Intuitively, DBSCAN finds dense regions that are separated by regions of lower density. The density of a region (given a distance function between pairs of data points) is defined by two parameters: the number of neighbors, *minPts*, and the radius, ϵ, of the neighborhood. A data point is called *core point* (i.e., it is at the core of a dense region) if it has at least minPts neighbors (including itself) within radius ϵ; a non-core point in the ϵ-neighborhood of a core point is a *border point* (i.e., it is at the border of a dense region); all other points are *noise* [100].

The runtime of the DBSCAN algorithm heavily depends on the efficiency of the neighborhood computation. In our experiments, the neighborhood computation accounts for up to 99% of the overall runtime for some datasets. Therefore, in order to efficiently cluster large collections of sets, effective indexing techniques for sets are required.

Similarity indexes for sets have been proposed in the context of ϵ-neighborhood joins, which are executed in an index nested loop fashion. A prominent representative is the *prefix index* [6, 29], which is linear in size and has been shown to be highly effective [81]. The *symmetric* prefix index returns the complete ϵ-neighborhood for a given query point r. The *asymmetric* prefix index assumes a processing order on the sets in R and retrieves only the *lookahead neighbors*: the ϵ-neighbors that follow r in processing order. A typical processing order for sets is based on the set sizes (ties broken arbitrarily). The *asymmetric* prefix index further leverages the length information to avoid many of the candidates that the symmetric index must inspect (among the unprocessed sets). Figure 3.1 illustrates the ϵ-neighborhood, the lookahead neighbors, and the candidate regions of symmetric and asymmetric index, respectively. The region above the gray line represents the sets that have been processed before r, the region below the gray line are unprocessed sets. The circles and semicircles show subset relationships.

Clearly, the asymmetric prefix index is preferable in terms of effectiveness over its symmetric counterpart. Unfortunately, there is an inherent mismatch between the asymmetric index and the DBSCAN algorithm. DBSCAN suffers from the following issues when executed with the asymmetric index: (1) *Core status problem:* The lookahead neighbors of r are not sufficient to update the core status of r. (2) *Border vs. Noise problem:* To distinguish border points from noise, a border point must be visible from a core point, which is not guaranteed by the asymmetric lookahead neighborhood. (3) *Disconnected clusters:* To guarantee connected clusters, DBSCAN imposes a (partial)

2 Hamming distance $H(r,s) = |r \cup s| - |r \cap s|$ for two sets r and s.

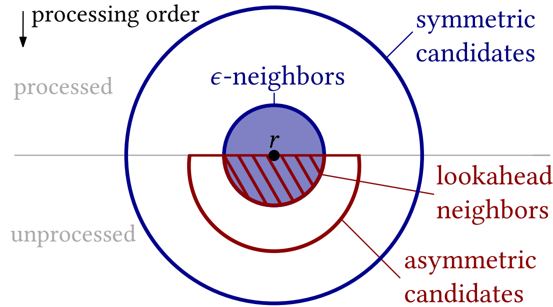


Figure 3.1: Symmetric candidates with ϵ-neighbors (blue); asymmetric candidates with lookahead neighbors (red).

processing order on the neighborhood computations: all core points of the current cluster must be expanded (i.e., their neighborhood must be computed) before any point belonging to a different cluster is processed.

A well-known clustering approach [20] is based on a self-join that precomputes and materializes all neighborhoods. The precomputed neighborhoods are then used while executing DBSCAN. This approach can leverage the asymmetric index and is efficient in runtime. Unfortunately, this join-based technique requires quadratic memory in the worst case and suffers from a large memory footprint in practice. For example, for our social media dataset (LIVEJ) that stores the interests of 3.1M users, this approach requires almost 100GB of memory.

Summarizing, applications that must cluster large collections of sets have two options, which we call Sym-Clust and Join-Clust. (1) Sym-Clust: Retrieve the full ϵ-neighborhoods in the processing order imposed by DBSCAN using the symmetric index. (2) Join-Clust: Compute and materialize neighborhoods in a join using the effective asymmetric index. None of the options is satisfying: Sym-Clust runs almost up to an order of magnitude slower than Join-Clust, while Join-Clust is infeasible for many datasets and parameter settings due to its excessive memory usage.

We propose a new clustering algorithm, *Spread*, that computes correct DBSCAN clusters using the asymmetric prefix index. Spread runs in linear space and does not need to materialize the (quadratic-size) neighborhoods. Spread avoids symmetric neighbor computations, therefore reducing the number of neighbors retrieved by Sym-Clust. So-called *backlinks* are introduced to achieve a correct clustering. Backlinks are dynamically added and removed as required and occupy only a small fraction of the memory that is used by materialized neighborhoods. Spread maintains a graph of subclusters in a disjoint-set data structure and guarantees that connected components in the resulting graph represent correct DBSCAN clusters.

In general, Spread can process data points in any *user-defined order* given an index that retrieves the lookahead neighbors, i.e., all data points that follow the query point in the user-defined processing order. In our usage scenario – set clustering – the processing order is defined by the set sizes (ties broken arbitrarily) and the asymmetric prefix index retrieves lookahead neighbors.

Summarizing, our contributions are the following:

- We propose *Spread*, a novel algorithm for partitioning large collections of sets into DBSCAN clusters. To the best of our knowledge, this is the first linear space DBSCAN-compliant algorithm that leverages the *asymmetric* prefix index for sets.
- We introduce the new concept of *backlinks* that keep sufficient information to build correct clusters independently of the processing order that the user imposes on Spread. We prove invariants for backlinks and the correctness of our approach.
- Our extensive empirical evaluation on 13 real-world datasets suggests that Spread is as fast as Join-Clust (that materializes all neighborhoods) while being competitive in memory usage with Sym-Clust (that computes all neighborhoods on the fly).

The remainder of this chapter is organized as follows. In Section 3.2, we cover the background on ϵ-neighborhood and set similarity, indexing techniques for sets, and density-based clustering, and we define the *density-based set clustering problem.* Section 3.3 presents the two baseline approaches for density-based set clustering, Join-Clust and Sym-Clust. In Section 3.4, we present Spread, our time- and space-efficient solution for density-based set clustering. We evaluate our solution against the baseline algorithms and discuss the results in Section 3.5. Related work is summarized in Section 3.6. Finally, Section 3.7 concludes this chapter.

3.2 BACKGROUND & PROBLEM DEFINITION

We revisit set similarity indexes and density-based clustering, and define our problem. To simplify the presentation, we focus on prefix indexes for the Hamming distance. Our results, however, extend to other distance and similarity measures (e.g., Jaccard or Cosine) and the respective indexes [37, 81, 108]. The required adaptations of the index that have been studied in the context of set similarity joins [81, 130] (e.g., prefix length, size lower bound, equivalent overlap) also apply to our scenario.

3.2.1 *Set Similarity and ϵ-Neighborhood*

R is a collection of n unique sets, each set $r \in R$ consists of unique tokens $t_1, \ldots, t_m$, $|r| = m$. The *processing order*, $\succ$, is a total order defined over R. The similarity between two sets r and s is assessed by the Hamming distance, $H(r, s) = |r \cup s| - |r \cap s|$, which counts the number of tokens that exist only in one of the sets, e.g., $H(r_1, r_2) = 4$ and $H(r_2, r_3) = 3$ for the sets in Figure 3.2.

The *ϵ-neighborhood* of set r includes r and all sets within distance ϵ from r, $N_\epsilon(r) = \{s \in R \mid H(r, s) \leq \epsilon\}$. A *region query* on r computes $N_\epsilon(r)$. A set r splits its ϵ-neighborhood into two disjoint parts based on the processing order: the *lookahead neighbors* that follow r in processing order, $N_\epsilon^{\succ}(r) = \{s \in N_\epsilon(r) \mid s \succ r\}$ and the *preceding neighbors*, $N_\epsilon^{\prec}(r) = \{s \in N_\epsilon(r) \mid s \prec r\}$.

3.2.2 *Indexing Techniques for Sets*

PREFIX FILTER AND INVERTED INDEX A naive approach computes a region query $N_\epsilon(r)$ by verifying the predicate $H(r,s) \leq \epsilon$ for all sets $s \in R$. An effective indexing technique, which was originally developed for set similarity joins [16, 81], is based on the so-called prefix filter. The prefix, π_r, of a set r consists of the first π tokens of r according to some total token order (which must be the same for all sets). The prefix length depends on the distance function and is $\pi = \epsilon + 1$ for the Hamming distance. Figure 3.2 shows the prefix of three sets for distance threshold $\epsilon = 3$ and a numerical token order. The prefix filter works best if the tokens in the prefix are infrequent, thus the tokens are typically ordered by ascending global token frequency.

A set $s \in R$ can be in the ϵ-neighborhood $N_\epsilon(r)$ only if the prefixes of r and s share at least one token, i.e., $H(r,s) \leq \epsilon \Rightarrow \pi_r \cap \pi_s \neq \emptyset$ (assuming $|r| + |s| > \epsilon$; otherwise r and s are always similar). Therefore, if two sets do not share a token in the prefix, the pair can be safely pruned. If two sets r and s share a prefix token, (r,s) is a *candidate pair* and must undergo *verification*, i.e., the predicate $H(r,s) \leq \epsilon$ must be evaluated. Candidates that fail verification are *false positives*. Mann et al. [81] discuss efficient prefix-based verification.

SYMMETRIC PREFIX INDEX An inverted index on the prefix tokens is used to retrieve candidate pairs efficiently. The inverted index maps prefix tokens to sets that contain that token in the prefix. A lookup of set r retrieves all lists of the prefix tokens of r. The union of these lists (except r itself) are the candidates of r. The index is *symmetric* and returns the ϵ-neighborhood of r.

For example, the candidates for r_2 returned by the symmetric index, $\pi = \epsilon + 1 = 4$, in Figure 3.2 are $\{r_1, r_3\}$ (resulting from the union of $[r_2, r_3]$ for token 1, $[r_2, r_3]$ for token 2, $[r_1, r_2]$ for token 5, and $[r_1, r_2]$ for token 6). Candidate r_1 is a false positive since $H(r_1, r_2) > \epsilon$; r_3 is a true positive due to $H(r_2, r_3) \leq \epsilon$.

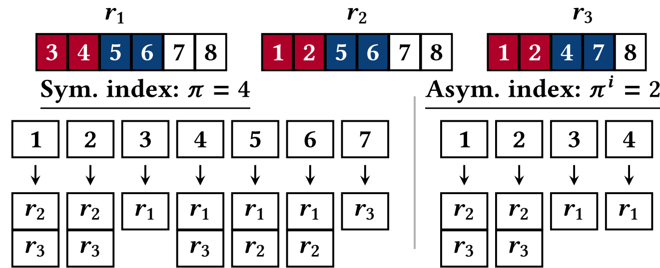


Figure 3.2: Symmetric and asymmetric prefix index, $\epsilon = 3$.

ASYMMETRIC PREFIX INDEX We construct an *asymmetric* prefix index that returns only the lookahead neighbors, $N_\epsilon^\succ(r)$. To this end, we define a length-based processing order on R (longest to shortest): r precedes s if $|r| > |s|$, i.e., $|r| > |s| \Rightarrow s \succ r$; ties ($|r| = |s|$) are broken by the lexicographic order of the sorted sets.

Since we are interested only in sets $s \in N_\epsilon(r)$ that are no larger than r, $|s| \leq |r|$, we need to index only a subset of the prefix tokens: the tokens in the so-called *indexing prefix* [130]. The so-called *probing prefix* of the lookup set, r, remains of length $\pi = \epsilon + 1$. For the Hamming distance, the indexing prefix is of length $\pi^i = \lfloor \frac{\epsilon}{2} + 1 \rfloor$. For $\epsilon > 0$, the probing prefix is always longer than the indexing prefix, e.g., $\pi = 4$ and $\pi^i = 2$ for $\epsilon = 3$. A shorter prefix results in fewer candidates and renders the asymmetric index more effective than its symmetric counterpart.

In the case of r_2, the asymmetric index, $\pi^i = 2$, in Figure 3.2 returns only a true positive candidate, r_3. The false positive candidate, r_1, which is returned by the symmetric index, is avoided.

3.2.3 *Density-Based Clustering*

We formally define DBSCAN clusters and the related concepts. A set r represents a point to be clustered. The *density* of r is the number of ϵ-neighbors $|N_\epsilon(r)|$ (cf. Section 3.2.1).

Core, Border, Noise. A set r is a *core point* iff the ϵ-neighborhood of r contains at least minPts sets: r is core $\Leftrightarrow |N_\epsilon(r)| \geq \text{minPts}$. A set s is a *border point* iff it is in the ϵ-neighborhood of a core point r and s is not core: s is border $\Leftrightarrow s \in N_\epsilon(r) \wedge |N_\epsilon(r)| \geq \text{minPts} \wedge |N_\epsilon(s)| < \text{minPts}$. All remaining sets in R are *noise*. We denote the set of core and border points with $\mathcal{C}$ and $\mathcal{B}$, respectively. The set of noise points is $\mathcal{N} = R \setminus (\mathcal{C} \cup \mathcal{B})$.

Density-Reachability. Let $r, s \in R$ and r is core: s is *directly density-reachable* from r iff s is in the ϵ-neighborhood of r: $r \blacktriangleright s \Leftrightarrow s \in N_\epsilon(r)$. If there is a sequence of sets $r_1, r_2, \ldots, r_k$ with $r_1 = r$ and $r_k = s$, $r_i \blacktriangleright r_{i+1}$ for $1 \leq i < k$, s is *density-reachable* from r, denoted $r \blacktriangleright \ldots \blacktriangleright s$. Two sets r, s are *density-connected* if there is a set x s.t. both r and s are density-reachable from x.

A *density-based cluster* is a subset $C_i \subseteq R$ that satisfies two criteria [104]:

1. *Maximality* $\mathbb{M}$*:* For any two sets $r, s \in R$, $r \in C_i$. If s is density-reachable from r, then $s \in C_i$. Formally,

$$\forall r, s \in R : r \in C_i \wedge r \blacktriangleright \ldots \blacktriangleright s \implies s \in C_i$$

2. *Connectivity* $\mathbb{C}$*:* For any two sets r, s in C_i, there is a set x that density-connects r and s. Formally,

$$\forall r, s \in R : r, s \in C_i \implies \exists x \in C_i : r \blacktriangleleft \ldots \blacktriangleleft x \blacktriangleright \ldots \blacktriangleright s$$

DBSCAN CLUSTERING A border point may be part of multiple density-based clusters such that the clusters overlap. We define the *DBSCAN clustering* that partitions the data into non-overlapping clusters. The standard DBSCAN algorithm [40] produces a DBSCAN clustering.

Definition 3.2.1. *Let $R^* = R \setminus \mathcal{N}$ and $C_1, C_2, \ldots, C_k$ be density-based clusters such that $\bigcup_{i=1}^k C_i = R^*$. A DBSCAN clustering is a partitioning $\Gamma = \{C'_1, C'_2, \ldots, C'_k\}$, $C'_i \subseteq C_i$, such that $\bigcup_{i=1}^k C'_i = R^*$, $C'_i \cap C'_j = \emptyset$ for $i \neq j$.*

A *subclustering* of a cluster C_i, $\psi_i = \{c_1, c_2, \ldots, c_l\}$, is a partitioning of C_i into $1 \leq l \leq |C_i|$ non-empty, disjoint subclusters, $c_j \subseteq C_i$, such that $\bigcup_{j=1}^{l} c_j = C_i$, $c_j \cap c_k = \emptyset$ for $j \neq k$.

A *subcluster graph* of R^* is an undirected graph in which nodes are subclusters and an edge between two nodes can only exist if the respective nodes are in the same DBSCAN cluster.

3.2.4 *The DBSCAN Algorithm*

The standard DBSCAN algorithm [40] forms clusters by repeatedly picking a seed point from the set of unvisited data points (initially all points are unvisited). If the seed is a core point, it forms a new cluster with all points that are density-reachable from the seed and are not yet assigned to a cluster. The set of density-reachable points is computed by recursively adding the ϵ-neighbors of all core points to the current cluster. The algorithm terminates when all points have been visited. Points that cannot be assigned to a cluster are noise.

Table 3.1: Notation overview.

Notation	Description
R	a collection of sets
r, s, x	sets of R
$\|r\|$	cardinality of set r
$r \prec s, r \succ s$	r precedes/succeeds s (in R)
$H(r, s)$	the Hamming distance of two sets r, s
π, π^i	probing/indexing prefix
ϵ	distance threshold
minPts	minimum density s.t. a set r is core
$N_\epsilon(r)$	full ϵ-neighborhood of r
$N_\epsilon^{\prec}(r), N_\epsilon^{\succ}(r)$	preceding/lookahead neighbors of r
$r \blacktriangleright s$	s is directly density-reachable from r
$r \blacktriangleright \ldots \blacktriangleright s$	s is density-reachable from r
$\mathcal{C}, \mathcal{B}, \mathcal{N}$	the set of core, border, and noise sets
C_i	a density-based cluster with id i

3.2.5 *Problem Statement*

Definition 3.2.2 (Density-Based Set Clustering)**.** *Given a collection of sets R, a distance threshold ϵ, and the neighborhood density minPts, the goal is to find a DBSCAN clustering $\Gamma = \{C_1, C_2, \ldots, C_k\}$ of R.*

For sets, asymmetric indexes with a lookahead neighbor function $N_\epsilon^{\succ}(r)$ promise the best performance (cf. Section 3.2.2). Given an ordering $\succ$ on R, we strive for a time- and space-efficient algorithm that solves the density-based set clustering problem with an asymmetric index.

RUNNING EXAMPLE Figure 3.3 shows an example collection R of ten sets, r_1–r_{10}, and their neighborhoods for Hamming distance $\epsilon = 3$. Sets r_3, r_5, r_6, and r_{10}, are core sets; all sets r_1–r_{10} form a single cluster.

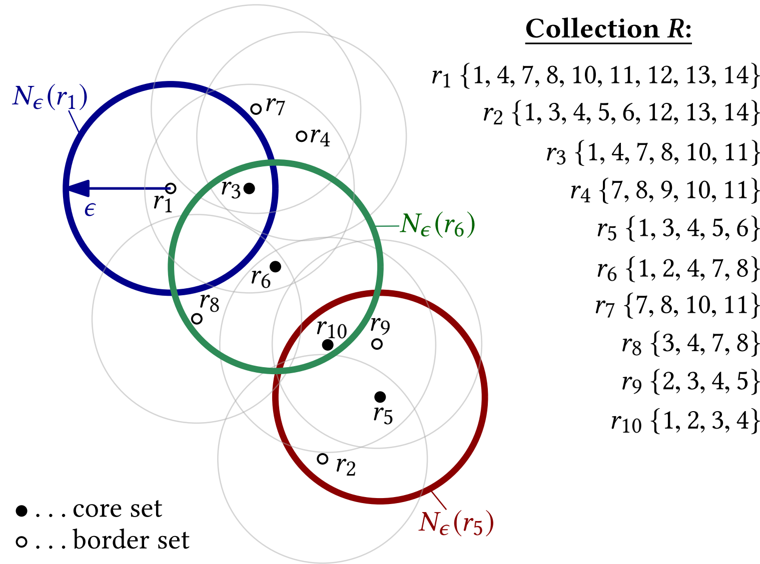


Figure 3.3: Running example, $\epsilon = 3$, minPts = 4.

3.3 BASELINE APPROACHES

This section presents two baseline solutions for the density-based set clustering problem. (1) *Sym-Clust* is memory-efficient and follows the standard DBSCAN approach with the symmetric prefix index to answer neighborhood queries on the fly. (2) *Join-Clust* is speed-optimized and materializes all ϵ-neighborhoods using a state-of-the-art set similarity join algorithm [16] (which leverages the asymmetric prefix index) before the standard DBSCAN algorithm is executed.

Both baselines leverage state-of-the-art set indexes. We are not aware of other previous solutions that can outperform Sym-Clust or Join-Clust for the density-based set clustering problem. Note that using the standard DBSCAN [40] (rather than some advanced techniques presented in later works, cf. Section 3.6) is not a limiting factor: Most of the overall execution time is spent computing the neighborhoods, and prefix-based indexes are highly efficient in combination with efficient verification [81].

3.3.1 *Sym-Clust: DBSCAN with Inverted Index*

When the standard DBSCAN algorithm (cf. Section 3.2.4) picks a seed point that is core, it forms a cluster with all points that are density-reachable from the seed. The density-reachable points are computed by pushing all core neighbors of the seed onto a stack. Then, each point on the stack is processed in the same manner (i.e., all its core

neighbors are pushed onto the stack) until the stack is empty. All neighbors of core points retrieved in this process belong to the cluster.

The neighborhood queries will overlap to some extent. Assume r is processed before s, $s \in N_\epsilon(r)$, then $|N_\epsilon(s) \cap N_\epsilon(r)| \geq 2$ (at least r and s are in both neighborhoods). Since r assigns all its neighbors to the current cluster, only the non-overlapping neighbors of s, $N_\epsilon(s) \setminus N_\epsilon(r)$, will further increase the cluster.

Figure 3.4 illustrates this observation for the neighborhoods of two example points r (black circle) and s (red circle): only the new, non-overlapping area of $N_\epsilon(s)$ (shaded in red) is relevant for expanding the cluster.

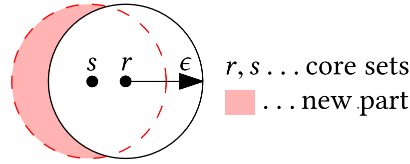


Figure 3.4: Redundant neighborhood queries.

The standard DBSCAN algorithm requires the use of a symmetric index since it assumes to see all neighbors of a point s when s is processed. The asymmetric index is not compatible with the standard DBSCAN algorithm: We cannot impose an order on the points such that all non-overlapping neighbors of s follow s in the processing order. Further, the size of the neighborhood of s, $|N_\epsilon(s)|$, is required to decide its core status.

Figure 3.5 shows the symmetric prefix index for our running example, $\epsilon = 3$, $\pi = \epsilon + 1 = 4$. We probe $r_8 = \{3, 4, 7, 8\}$. The prefix of r_8 consists of all tokens in r_8 (due to $|r_8| = \pi$). The union of the respective index lists yields the candidates $\{r_2, r_5, r_9, r_{10}, r_1, r_3, r_6, r_4, r_7\}$. Note that the candidates include both sets that are smaller and sets that are larger than r_8.

The so-called *length filter* [16], an optimization of the symmetric prefix index that also applies to its asymmetric counterpart, prunes candidates r_2 and r_1. Due to their length difference to r_8, they cannot be in the ϵ-neighborhood of r_8. By ordering the lists in processing order (i.e., longer sets precede shorter sets, as illustrated in Figure 3.5), the length filter can prune the head (sets that are too long) and the tail (sets that are too short) of a list without inspecting all elements in head and tail, respectively.

All candidates that are not pruned by the length filter must undergo verification. Only r_6 passes verification, therefore $N_\epsilon(r_8) = \{r_8, r_6\}$, and r_8 is classified as non-core (minPts = 4).

COMPLEXITY ANALYSIS We probe each set $r \in R$ against the index once. With cost C for an index lookup and $n = |R|$, the runtime is $O(n \cdot C)$; $C = O(n)$ since r may have $O(n)$ neighbors, thus the overall runtime of Sym-Clust is $O(n^2)$. The symmetric index is of linear size leading to space complexity $O(n)$.

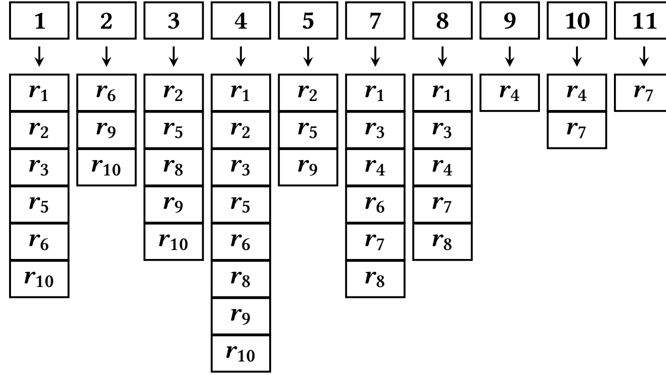


Figure 3.5: Symmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi = 4$.

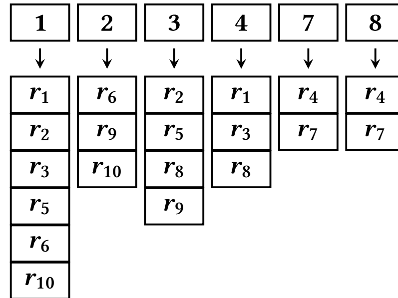


Figure 3.6: Asymmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi^i = 2$.

3.3.2 *Join-Clust: Materialized Neighborhoods*

Join-Clust executes a set similarity self-join on R and materializes the ϵ-neighborhoods in main memory. The self-join traverses all sets of $r \in R$ in processing order and computes their lookahead neighbors, $N_\epsilon^{\succ}(r)$. The lookahead neighbors of r are appended to the list of r's neighbors, and r is appended to the neighborhood lists of all $s \in N_\epsilon^{\succ}(r)$. After processing all sets, the neighborhood list of each set $r \in R$ is complete and stores $N_\epsilon(r)$.

Next, standard DBSCAN (cf. Section 3.2.4) is executed to form clusters using the materialized neighborhoods. Algorithms 9–12 implement the similarity join with neighborhood materialization, index creation, probing, and efficient verification [81].

Mann et al. [81] found that the prefix-based index in combination with the length filter can be considered state of the art given an efficient verification procedure (which we use).

Figure 3.6 shows the asymmetric prefix index for our running example, $\epsilon = 3$, $\pi^i = 2$. We probe $r_8 = \{3, 4, 7, 8\}$ and look up the lists of the tokens 3, 4, 7, and 8 (the length of the probing prefix is $\pi = 4$). Since we are only interested in the lookahead neighbors, i.e., all neighbors that follow r_8 in processing order, we need to inspect the lists only starting from the point where r_8 or a set $r_i \succ r_8$ appears. The length filter does not prune any candidate in this example, and the candidate set is $\{r_9\}$. Since $H(r_8, r_9) > \epsilon$, the lookahead neighborhood of r_8 is empty, $N_\epsilon^{\succ}(r_8) = \emptyset$.

Algorithm 9: Materialize-Neighborhoods(R, ϵ)

```
   Input: Collection of sets R, distance threshold ε
   Result: Materialized neighborhoods of R w.r.t. ε
1  I ← Create-Index (R, ε);
2  pairs ← ∅ // Result set of similar pairs
3  foreach r ∈ R in processing order do
4      M ← Probe (r, I, ε) // candidates with prefix overlaps
5      foreach (s, po) ∈ M do // po . . . prefix overlap
6          if Verify-Pair (r, s, ε, po) then
7              pairs ← pairs ∪ {(r, s)}
8  neighborhoods ← new associative array of size |R|;
9  foreach (r, s) ∈ pairs do
10     neighborhoods[r] ← neighborhoods[r] ∪ {s};
11     neighborhoods[s] ← neighborhoods[s] ∪ {r};
12 return neighborhoods
```

Algorithm 10: Create-Index (R, ϵ)

```
   Input: Collection of sets R, distance threshold ε
   Result: Inverted index of R w.r.t. ε
1  I ← ∅ // inv. index of set prefixes, I_r[p] . . . list of token r[p]
2  foreach r ∈ R do
3      π ← ⌊(ε+2)/2⌋ // indexing prefix length of r
4      for p ← 1 to π do I_r[p] ← I_r[p] ∪ {r};
5  return I
```

Algorithm 11: Probe (r, I, ϵ)

```
   Input: Probing set r, inv. index I, distance threshold ε
   Result: Set of candidates for r w.r.t. ε
   // M maps a candidate s to its prefix overlap with r
1  M ← new associative array // candidates
2  π ← ε + 1 // probing prefix length of r
3  lb_r ← |r| − ε // size lower bound wrt. r
4  for p ← 1 to π do
5      foreach s ∈ I_r[p] in proc. order do // list of token r[p]
6          if |s| < lb_r then break ;
7          else // add candidate
8              if s ∉ M then M[s] ← 0; // init.
9              M[s] ← M[s] + 1 // incr. overlap of (r, s)
10 return M
```

Algorithm 12: Verify-Pair (r, s, ϵ, po)

Input: Probing set r, candidate set s, distance threshold ϵ, prefix overlap po
Result: True iff r and s are similar w.r.t. ϵ, false otherwise

```
// cf. Mann et al. [81] for prefixes, equiv. overlaps, and Verify proc.
```

1 $\pi_r, \pi_s \leftarrow$ probing resp. indexing prefix length of r resp. s;
2 $w_r, w_s \leftarrow \pi_r$- resp. π_s-th token in r resp. s;
3 $t \leftarrow$ equivalent overlap for r, s, and ϵ;
4 **if** $w_r < w_s$ **then**
5 | **return** Verify $(r, s, t, po, \pi_r + 1, po + 1)$
6 **return** Verify $(r, s, t, po, po + 1, \pi_s + 1)$

In the context of the self-join, r_8 will be retrieved as a lookahead neighbor of r_6, which is processed before r_8. Therefore, the neighborhood list of r_6 will store r_8 and vice versa.

Join-Clust produces fewer candidates than Sym-Clust and is therefore faster. However, the efficiency of Join-Clust comes at the cost of a larger memory footprint since all neighborhoods must be materialized.

COMPLEXITY ANALYSIS A neighborhood query is a constant-time lookup and a traversal of $O(|N_\epsilon(r)|)$ neighbors. In the worst case, the join reports $O(n^2)$ pairs. Consequently, materializing the neighborhoods takes $O(n^2)$ time and space for $n = |R|$. The asymmetric prefix index requires only $O(n)$ space and does not dominate memory usage.

3.4 THE SPREAD ALGORITHM

We present *Spread*, a novel time- and space-efficient solution for the density-based clustering problem. Spread leverages the effective asymmetric index and clusters all sets by traversing the sets in processing order. We identify key challenges that must be solved, discuss the algorithm, prove its correctness, analyze time and space complexity, and sketch a multi-core extension.

3.4.1 *Key Challenges*

Since Spread uses an asymmetric neighborhood index, a processing order, $\succ$, must be imposed on the data points, and an index lookup of query point r retrieves only the lookahead neighbors, $N_\epsilon^\succ(r)$. To achieve a correct clustering without materializing the neighborhoods, three key challenges must be solved.

In the following discussion, we assume that all sets of R are processed in processing order. When the *current set* r_i is to be processed, we know the core status of all preceding sets, $r_j \prec r_i$, but we do not know the core status of any unprocessed sets, $r_k \succ r_i$. We further assume that all sets $r \in R$ that are directly density-reachable from any r_j that precedes r_i (i.e., are neighbors of a core point $r_j \prec r_i$) are assigned to the same cluster as the core point r_j; this may also include sets $r_k \succ r_i$ that have not been processed yet.

CORE STATUS A set r_i is core if $|N_\epsilon(r_i)| \geq$ minPts. Sym-Clust and Join-Clust have access to the full neighborhood, $N_\epsilon(r_i)$, thus deciding the core status of r_i is trivial. In contrast, Spread sees only the lookahead neighbors, $N_\epsilon^\succ(r_i)$. To identify the core status of r_i, however, additional knowledge about the size of the preceding neighborhood, $N_\epsilon^\prec(r_i)$, is required.

Consider probing $r_i = r_5$ in our running example. According to our assumptions, the core status of sets r_1–r_4 is known (only r_3 is core), and all neighbors of r_3 are in cluster $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$. An index lookup of r_5 returns $N_\epsilon^\succ(r_5) = \{r_9, r_{10}\}$. Since $|N_\epsilon^\succ(r_5)| + 1 = 3 < 4 =$ minPts we cannot decide if r_5 is core. In fact, r_5 should be classified core since the full neighborhood is $N_\epsilon(r_5) = \{r_2, r_5, r_9, r_{10}\}$ (cf. red circle in Figure 3.3).

BORDER VS. NOISE Assume that the current set r_i is a non-core point that is not assigned to any cluster. We need to decide if r_i is border or noise. A border point has at least one core point in its neighborhood. None of the preceding neighbors, $r_j \in N_\epsilon^\prec(r_i)$, is core, otherwise r_i would be assigned to the cluster of r_j. Thus, r_i is core iff one of the lookahead neighbors is core. Unfortunately, we do not know the core status of the lookahead neighbors and can therefore not label r_i as border or noise.

Assume a core point, $r_k \in N_\epsilon^\succ(r_i)$, among the lookahead neighbors of r_i. When r_k is processed, r_k will not see r_i in its lookahead neighborhood since $r_i \prec r_k$. Therefore, r_i will not be included into the cluster of r_k and will wrongly be classified noise. The challenge is to correctly decide the border status of r_i despite seeing only the lookahead neighbors of r_i and r_k.

In our running example, r_1 is processed first. Thus, no core points are known and no clusters exist. $N_\epsilon^\succ(r_1) = \{r_3\}$ and r_1 remains noise (cf. blue circle in Figure 3.3). When the neighbor r_3 of r_1 is processed, r_3 will be detected as a core point and start a new cluster. However, since r_3 only sees its lookahead neighbors, $N_\epsilon^\succ(r_3) = \{r_4, r_6, r_7\}$, r_1 is not included into the cluster and is not detected as a border point.

DISCONNECTED CLUSTERS. Assume that the current set r_i is core and there is a core point $r_j \prec r_i$ in a cluster C_j, $r_i \notin C_j$. The current set r_i will assign all its lookahead neighbors to its cluster, $C_i = C_i \cup N_\epsilon^\succ(r_i)$ (C_i can be a new cluster started by r_i or an existing cluster to which r_i belongs). Unfortunately, we cannot assume that C_i and C_j are indeed distinct clusters: there can be a core point $r_k \succ r_i$ that density-reaches both r_i and r_j, i.e., C_i and C_j should form a single cluster. In general, multiple subclusters of the same DBSCAN cluster may grow independently. The challenge is to identify subclusters that should be merged and to merge them efficiently.

We process the current set $r_i = r_6$ in our running example. According to our assumptions, we know that r_3 and r_5 are core and we are aware of two clusters, $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$, $C_5 = \{r_2, r_5, r_9, r_{10}\}$. In addition, assume that we know that r_6 is core. Then, r_6 extends its current cluster, C_3, with its lookahead neighbors $N_\epsilon^\succ(r_6) = \{r_8, r_{10}\}$. Note that r_{10} is already part of cluster C_5. Since we do not know the core status of r_{10}, we cannot decide if C_5 and C_3 should be merged into a single cluster. If r_{10} is core, r_5 and r_3 are density-reachable from r_{10} and should be in the same cluster. If r_{10} is a

border point, however, the clusters must not be merged, and r_{10} can be assigned to either C_5 or C_3.

3.4.2 *Data Structures*

DISJOINT-SET The disjoint-set (or union-find) data structure maintains a dynamic collection of non-overlapping sets for n objects in $\mathcal{O}(n)$ space [33, 112]. A typical use case is the efficient computation of (minimum) spanning trees. It supports three operations: (1) For a given element u, make_set (u) creates a new (singleton) set that contains u. (2) The union (u, v) operation merges the two sets that contain u resp. v into a new set. (3) find_set (u) returns the representative for the set that contains u or ∞ if u is not found. The amortized worst-case time complexity is $\Theta(\alpha(n))$ for all operations, $\alpha(.)$ being the inverse Ackermann function. In practice, $\alpha(n)$ is considered a constant. In our setting, set elements are subclusters, and the disjoint-set data structure links subclusters that belong to the same DBSCAN cluster.

BACKLINKS The backlinks data structure of a set $r \in R$ is a collection of unique references to other sets s that precede r, $s \prec r$. The backlinks bl support the add operation, $bl \cup \{s\}$, which adds a reference to a new set s in time $\mathcal{O}(1)$ (on average). Depending on the type of sets that are referenced in the backlinks, we distinguish core and non-core backlinks, denoted *c_bl* and *nc_bl*, respectively. We implement backlinks as unordered sets of integer identifiers.

3.4.3 *The Algorithm*

Algorithm 13 shows the pseudocode of Spread. We use the following notation: r is the current probing set, $s \succ r$ is a lookahead neighbor, and $x \prec r$ is a preceding neighbor. Initially, all sets are noise, i.e., their cluster identifier is $-\infty$, $\forall r \in R : r.cid = -\infty$. Although we initialize all sets in Algorithm 13 explicitly (lines 3–4), this can also be done during indexing (cf. Algorithm 10).

ALGORITHM OUTLINE Spread proceeds in three main steps: (1) A counter and the processing order guarantee that the cardinality of the ϵ-neighborhood is known when a set is processed despite using the asymmetric prefix index. (2) Each set is assigned to a subcluster solely based on its lookahead neighboorhood. Subclusters of the same DBSCAN cluster are linked in a subcluster graph. Backlinks ensure that we do not miss border sets or links between subclusters. (3) Each connected component in the subcluster graph represents a DBSCAN cluster.

CORE STATUS A set r is core if $|N_\epsilon(r)| \geq$ minPts. In Spread, however, only $N_\epsilon^{\succ}(r)$ is computed. To capture the cardinality of $N_\epsilon^{\prec}(r)$, we store a density counter with each set r, denoted $r.dens$. Initially, $\forall r \in R : r.dens = 1$. For every lookahead neighbor $s \in N_\epsilon^{\succ}(r)$, $r.dens$ and $s.dens$ are incremented (due to the symmetry of the distance). Core set identification is highlighted in green ☐.

BORDER VS. NOISE A probing set r that is not core is a border set iff $\exists\, y \in N_\epsilon(r)$: y is core. Due to our processing order and the fact that only $N_\epsilon^{\succ}(r)$ is computed, the existence of a core neighbor y may be unknown when r is probed. However, for each $s \in N_\epsilon^{\succ}(r)$, we know that r is part of $N_\epsilon^{\prec}(s)$. We store this information by adding r to the non-core backlinks $nc_bl[s]$ of each $s \in N_\epsilon^{\succ}(r)$ (lines 31–33). Then, the first $s \in N_\epsilon^{\succ}(r)$ that becomes core claims r (and all other unassigned sets in $nc_bl[s]$) as border point for its subcluster. If none of the neighbors $s \in N_\epsilon^{\succ}(r)$ becomes core, then r remains noise. Lines 26–30 deal with a special case: If any $s \in N_\epsilon^{\succ}(r)$ is already core when r is probed, then s claims r immediately without adding r to its non-core backlinks. The relevant code lines are marked in red ■.

SUBCLUSTER LINKAGE If the probing set r is core and a core neighbor y is part of another subcluster, the subclusters of r and y must be linked in our subcluster graph. The subcluster graph represents all connected components of subclusters, each of which is a DBSCAN cluster. We use the disjoint-set data structure ds to track the connected components. Two subclusters u, v are linked by ds.union(u, v). We may not be able to determine if there is a set $s \in N_\epsilon^{\succ}(r)$ that is core before s is probed. We use the core backlinks, c_bl, to book-keep potential subclusters for linkage: r adds its subcluster identifier to $c_bl[s]$ of each $s \in N_\epsilon^{\succ}(r)$ (lines 22-23). After $N_\epsilon^{\succ}(r)$ has been processed, a link between the subcluster of r and every entry in $c_bl[r]$ is created (line 24). The special case when s is already core allows us to create the link immediately without using core backlinks (lines 20–21). Linkage is only required if two subclusters coalesce (condition in line 19). Otherwise, r simply claims $s \in N_\epsilon^{\succ}(r)$ for its subcluster (lines 17–18). Linkage of subclusters is highlighted in blue ■.

All backlinks of r are released after r has been processed to save memory (line 34). The subcluster graph in ds is used to assign consistent cluster IDs in a final scan over R (lines 35–36).

3.4.4 *Correctness*

We show that Algorithm 13 partitions R into DBSCAN clusters (cf. Definition 3.2.1). Set $r_i \in R$ is the i-th set of R in processing order. We prove the correctness by induction over i and increasing subsets $R^i \subseteq R$. $R^0 = \emptyset$, $R^i = R^{i-1} \cup \{r_i\} \cup N_\epsilon^{\succ}(r_i)$ for $1 \leq i \leq n = |R|$, thus $R^n = R$. In the following, we sketch the proofs of the invariants that must be shown.

CORE STATUS The core status of set r_i is determined in the i-th iteration of the main loop. r_i is core if minPts $\leq |N_\epsilon(r_i)| = 1 + \left|N_\epsilon^{\prec}(r_i)\right| + \left|N_\epsilon^{\succ}(r_i)\right|$. In line 5, $r_i.dens = 1 + \left|N_\epsilon^{\prec}(r_i)\right|$. Lines 6–11 compute $N_\epsilon^{\succ}(r_i)$. The index lookup in line 6 returns candidate set M, $N_\epsilon^{\succ}(r_i) \subseteq M \subseteq \{s \mid s \succ r_i\}$. Every set $s \in M$ is verified in line 9 such that $N_\epsilon^{\succ}(r_i)$ is available starting from line 12.

Algorithm 13: Spread(R, ϵ, minPts)

Input: Collection of sets R, distance threshold ϵ, min. density minPts

Result: A correct DBSCAN clustering of R w.r.t. ϵ, minPts

```
1  ds ← new disjoint-set; nc_bl, c_bl ← new backlinks;
2  I ← Create-Index (R, ϵ);
3  foreach r ∈ R do
4      r.dens ← 1; r.cid ← −∞; ds.make_set (r.id);
5  foreach r ∈ R in processing order do
6      M ← Probe (r, I, ϵ);
7      N_ϵ^≻ (r) ← ∅;
8      foreach (s, po) ∈ M do // po ... prefix overlap
9          if Verify-Pair (r, s, ϵ, po) then
10             r.dens ← r.dens + 1; s.dens ← s.dens + 1;
11             N_ϵ^≻ (r) ← N_ϵ^≻ (r) ∪ {s};
12     if r.dens ≥ minPts then // r is core
13         if r.cid = −∞ then r.cid ← r.id;
14         foreach x ∈ nc_bl[r] do // claim border sets x < r
15             if x.cid = −∞ then x.cid ← r.cid;
16         foreach s ∈ N_ϵ^≻ (r) do // s > r
17             if s.cid = −∞ then // claim unclaimed s > r
18                 s.cid ← r.cid
19             else if r.cid ≠ s.cid then // s already claimed
20                 if s.dens ≥ minPts then // s is core
21                     ds.union (r.cid, s.cid) // link subclusters
22                 else // remember core neighbor r
23                     c_bl[s] ← c_bl[s] ∪ {r.cid}
24         foreach x ∈ c_bl[r] do ds.union (r.cid, x);
25     else // r is not core, i.e., r.dens < minPts
26         if r.cid = −∞ then // claim potential border set r
27             foreach s ∈ N_ϵ^≻ (r) do
28                 if s.dens ≥ minPts then // s is core
29                     if s.cid = −∞ then s.cid ← s.id;
30                     r.cid ← s.cid; break;
31         if r.cid = −∞ then // remember potential border set r
32             foreach s ∈ N_ϵ^≻ (r) do
33                 nc_bl[s] ← nc_bl[s] ∪ {r}
34     release c_bl[r] and nc_bl[r] // not needed anymore
35 foreach r ∈ R do // final assignment of cluster IDs
36     if r.cid ≠ −∞ then r.cid ← ds.find_set (r.cid);
```

Lemma 3.4.1. *Algorithm 13 correctly identifies* all *core sets in R.*

Proof Sketch. We show that at the start of the i-th iteration in line 5, for all r_k and r_j, $1 \leq k < i \leq j$ the following invariants hold: (I1) $r_k.dens = |N_\epsilon(r_k)|$; (I2) $r_j.dens = 1 + \left|\{r_k \mid r_j \in N_\epsilon^{\succ}(r_k)\}\right|$, i.e., $r_i.dens = 1 + \left|N_\epsilon^{\prec}(r_i)\right|$. Further, (I3) in line 12 of the i-th iteration, $r_i.dens = |N_\epsilon(r_i)|$, i.e., Algorithm 13 correctly identifies the core status of r_i. □

BORDER VS. NOISE Lines 25–33 cover the case that r_i is not core. If any $s \in N_\epsilon^{\succ}(r_i)$ qualifies as core, s claims r_i. Otherwise, r_i is stored in the non-core backlinks $nc_bl[s]$ of every $s \in N_\epsilon^{\succ}(r_i)$ (lines 31–33). The next core neighbor in processing order claims r_i (lines 14–15) such that all border sets are assigned to a cluster.

Lemma 3.4.2. *Algorithm 13 correctly clusters* all *border sets in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all border sets $r_k \in \mathcal{B}$, $1 \leq k < i$: if r_k is not stored in $nc_bl[s]$ for any $s \in N_\epsilon^{\succ}(r_k)$, $s = r_i$ or $s \succ r_i$, then r_k is assigned to the cluster of a core point in its neighborhood. □

SUBCLUSTER LINKAGE Lines 12–24 cover the case that r_i is core. Each core point may form a subcluster on its own or together with other core points. We must ensure that all subclusters of the same DBSCAN cluster are linked in the disjoint-set, ds.

Lemma 3.4.3. *Algorithm 13 correctly links* all *subclusters in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all core neighbors $c \in CN(r_k) = N_\epsilon(r_k) \cap C$ of a core set $r_k \in C$, $1 \leq k < i$: (a) c and r_k have the same cluster representative (in ds), or (b) c is stored in some $c_bl[s]$, $s \in N_\epsilon^{\succ}(r_k)$, $s = r_i$ or $s \succ r_i$. □

Theorem 3.4.4. *Algorithm 13 returns a correct set clustering $\Gamma = \{C_1, C_2, \ldots, C_k\}$ of R according to Definition 3.2.1.*

Proof Sketch. By Lemmata 3.4.1–3.4.3 and due to our final scan over R (lines 35–36), $x.cid = ds.\text{find_set}(x.cid)$ holds for all $x \in R$. Initially, $x.cid = -\infty$ for all $x \in R$. The cluster IDs are updated only for border and core sets. Consequently, $x.cid = -\infty$ holds for all $x \in R \setminus (C \cup \mathcal{B}) \equiv \mathcal{N}$, i.e., also noise is correctly identified. □

3.4.5 *Complexity Analysis*

Memory. The asymmetric prefix index requires $O(n)$ space. In addition, Spread maintains the following data structures. (i) A density counter for each set $r \in R$ requires $O(n)$ space. (ii) A disjoint-set data structure with at most $O(n)$ entries, i.e., the disjoint-set structure requires $O(n)$ space [112]. (iii) In the worst case, we allocate two backlink structures for each $r \in R$, i.e., $O(n)$ backlinks. We release $c_bl[r]$ and $nc_bl[r]$ after probing r. Backlinks are only extended in lines 23 and 33. However, both lines are executed iff $\nexists s \in N_\epsilon^{\succ}(r) : s$ is core. Set s is core iff $s.dens \geq$ minPts, and the density is

updated for every neighbor, therefore any backlink holds at most minPts entries. As a result, no more than $O(n \cdot \text{minPts})$ entries are allocated, thus requiring $O(n)$ space since minPts and ϵ are constants. *Runtime.* For each $r \in R$, we process $O\left(\left|N_\epsilon^>(r)\right|\right)$ neighbors and the backlinks of r if it is core. Recall that the disjoint-set operations take constant time. Therefore, the final for-loop (lines 35–36) runs in $O(n)$ time. Overall, Spread runs in $O\left(n^2\right)$ time and $O(n)$ space.

3.5 EXPERIMENTAL EVALUATION

ALGORITHMS We compare our solution, Spread, against the two baseline approaches Sym-Clust and Join-Clust (cf. Section 3.3). All algorithms are single-threaded C++ implementations (2017 standard). Our implementations of Spread, Join-Clust, and the index of Sym-Clust follow the guidelines by Mann et al. [81], e.g., regarding symmetric and asymmetric prefix index, candidate generation, and optimized prefix-based verification.

DATASETS We execute all experiments on 13 real-world datasets: (a) Nine of the datasets where previously used for benchmarking set similarity joins [45, 81]: BMS-POS, DBLP, ENRON, FLICKR, KOSARAK, LIVEJ, NETFLIX, ORKUT, and SPOT. For a description of the datasets and preprocessing instructions[3] we refer to Mann et al. [81]. (b) Four large real-world datasets from the process mining domain, CELONIS1–4, that store one set per process. Compared to most datasets of the join benchmark, the universe size of these datasets is rather small. Table 3.2 summarizes important characteristics of our benchmark data.

Due to space constraints we omit detailed results for the following datasets: (a) DBLP, ENRON, and NETFLIX show very low runtimes (< 4s) and a small and stable memory footprint (< 1GiB) for all algorithms and configurations. (b) CELONIS3–4 show results similar to the other process mining datasets.

Table 3.2: Characteristics of datasets.

Dataset	Coll. Size	Set Size		Univ. Size
		avg.	max.	
BMS-POS[4]	$3.2 \cdot 10^5$	9.3	164	$1.7 \cdot 10^3$
FLICKR[5]	$1.2 \cdot 10^6$	10.1	102	$8.1 \cdot 10^5$
KOSARAK[6]	$6.1 \cdot 10^5$	11.9	$2.5 \cdot 10^3$	$4.1 \cdot 10^4$
LIVEJ[7]	$3.1 \cdot 10^6$	36.4	300	$7.5 \cdot 10^6$
ORKUT[7]	$2.7 \cdot 10^6$	119.7	$4.0 \cdot 10^4$	$8.7 \cdot 10^6$
SPOT[8]	$4.4 \cdot 10^5$	12.8	$1.2 \cdot 10^4$	$7.6 \cdot 10^5$
CELONIS1	$8.2 \cdot 10^6$	20.3	91	$1.2 \cdot 10^4$
CELONIS2	$2.6 \cdot 10^6$	22.1	130	$3.5 \cdot 10^3$

3 http://ssjoin.dbresearch.uni-salzburg.at/datasets.html

PARAMETERS The algorithms take two parameters: the neighborhood radius, ϵ, and the density, minPts. Typically, density-based clustering is sensitive to ϵ and quite robust to minPts. In our experiments, we vary both parameters: $\epsilon \in \{2, \mathbf{3}, 4, 5\}$ and minPts $\in \{2, 4, 8, \mathbf{16}, 32, 64, 128\}$ (defaults in bold font).

ENVIRONMENT All experiments have been conducted on a 64-bit machine with 2 physical Intel Xeon E5-2630 v3 CPUs, 2.40GHz, 8 cores each (i.e., 16 logical processors, hyper-threading disabled). The cores share a 20MiB L3 cache and have another 256KiB of independent L2 cache. The system has 96GiB of RAM and runs Debian 10 Buster (Linux 4.19.0-12-amd64 #1 SMP Debian 4.19.152-1 (2020-10-18)). Our code is compiled with `clang`[9] version 7, highest optimization level (`-O3`). The runtime is measured with `clock_gettime`[10] at process level, memory usage is the `heap peak` of Linux `memusage`[11] (using `LD_PRELOAD`). A single instance is executed at a time with no other load on the machine.

3.5.1 *Index & Cluster Statistics*

We compare the number of candidates, true positives, and the number of clusters. The numbers are sums over all region queries. Table 3.3 shows the results obtained for BMS-POS, KOSARAK, and CELONIS1. We observe that Spread produces exactly the same number of candidates as Join-Clust since both solutions use the asymmetric index. Sym-Clust generates significantly more candidates due to the symmetric prefix index and the symmetric distance computations. For CELONIS1, Spread and Join-Clust verify about 5 times fewer candidates compared to Sym-Clust.

3.5.2 *Runtime Efficiency*

We measure the overall runtime, i.e., the CPU time that is required to cluster all sets into DBSCAN clusters (excluding the time to load the data from disk). Figure 3.7 shows the results for varying ϵ (minPts = 16). We observe that Sym-Clust is not competitive in terms of overall runtime in most cases. For all datasets, except KOSARAK and SPOT, the runtime of Sym-Clust increases much faster with ϵ than observed for Join-Clust and Spread. This is mainly due to the use of the symmetric prefix index (more candidates) and redundant computations (symmetric pairs).

Our experiments reveal that Join-Clust suffers from the following issues: (i) High runtimes for LIVEJ, ORKUT, and SPOT due to the expensive neighborhood materializa-

4 BMS-POS: `http://www.kdd.org/kdd-cup/view/kdd-cup-2000` [138]
5 FLICKR: Bouros et al. [23]
6 KOSARAK: `http://fimi.uantwerpen.be/data/`
7 LIVEJ, ORKUT: `http://socialnetworks.mpi-sws.org/data-imc2007.html` [84]
8 SPOT: Pichl et al. [94]
9 `https://releases.llvm.org/7.0.0/tools/clang/docs/ReleaseNotes.html`
10 `https://man7.org/linux/man-pages/man2/clock_gettime.2.html`
11 `https://man7.org/linux/man-pages/man1/memusage.1.html`

Table 3.3: Index & cluster statistics for $\epsilon = 3$, minPts = 16.

(a) BMS-POS.

	Candidates	**True Positives**	**Clusters**
Sym-Clust	$3.9 \cdot 10^9$	$38.0 \cdot 10^6$	1
Join-Clust	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1
Spread	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1

(b) KOSARAK.

	Candidates	**True Positives**	**Clusters**
Sym-Clust	$40.7 \cdot 10^9$	$2.8 \cdot 10^9$	5
Join-Clust	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5
Spread	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5

(c) CELONIS1.

	Candidates	**True Positives**	**Clusters**
Sym-Clust	$644.6 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Join-Clust	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Spread	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075

tion. (ii) Join-Clust runs out of memory for many instances (missing points in plots), in particular for FLICKR (any ϵ), KOSARAK ($\epsilon \geq 4$), LIVEJ, ORKUT, and SPOT ($\epsilon \geq 3$).

Spread outperforms its competitors in most settings and is competitive with Join-Clust otherwise (cf. Figures 3.7a, 3.7g, and 3.7h). For CELONIS1 and CELONIS2, Spread outperforms Sym-Clust by almost an order of magnitude and is competitive with Join-Clust.

Figure 3.8 shows the runtime results for varying minPts values ($\epsilon = 3$). We observe that the runtime of all three solutions is quite robust to minPts. The insights are similar for all datasets and values of ϵ. We include the plots for BMS-POS and KOSARAK.

3.5.3 *Memory Efficiency*

We study the memory usage of Join-Clust, Sym-Clust, and Spread. All three solutions store (i) the collection, (ii) the inverted index, (iii) the candidates, and (iv) the result of a region query on the heap. The symmetric prefix index of Sym-Clust is larger than the asymmetric index, but still linear in the collection size. Sym-Clust generates more candidates than Join-Clust and Spread (cf. Section 3.5.1), which both use the asymmetric prefix index . Join-Clust materializes all neighborhoods in main memory. Sym-Clust and Spread materialize only a single neighborhood at a time. Spread stores also backlinks and the disjoint-set in main memory.

Figure 3.10 shows our results for varying ϵ (minPts = 16, y-axis log scale). Join-Clust runs out of memory for many instances (cf. Section 3.5.2). The neighborhood materialization in Join-Clust can be memory intensive even for small values of ϵ. We observe different growth rates with increasing radius ϵ, which we attribute to the different

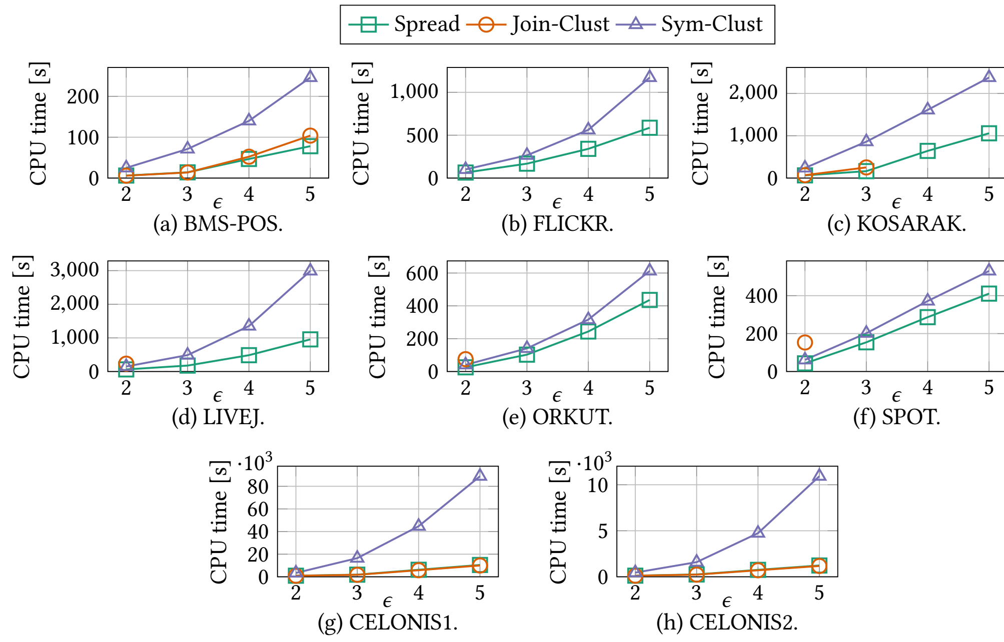


Figure 3.7: Runtime over ϵ, minPts = 16.

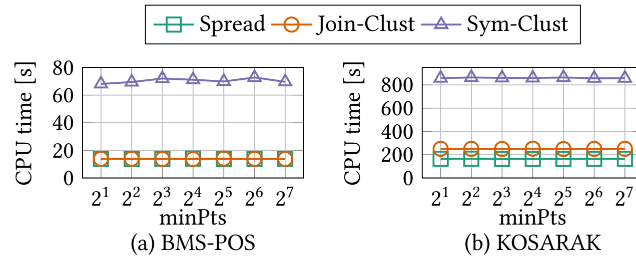


Figure 3.8: Runtime over minPts, ϵ = 3.

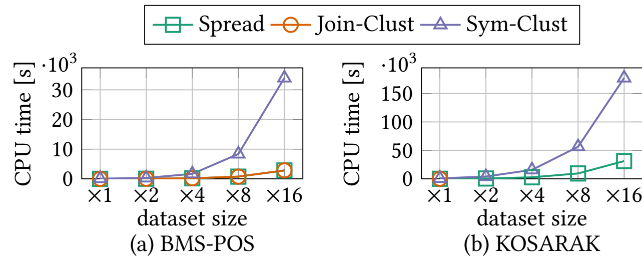


Figure 3.9: Runtime over data size, ϵ = 3, minPts = 16.

neighborhood sizes. The memory consumption of Sym-Clust is significantly lower and robust to varying ϵ. Spread shows a similar behavior. In some cases (e.g., LIVEJ, ORKUT), Spread occupies even less memory than Sym-Clust. When few backlinks are materialized, the smaller asymmetric prefix index of Spread outweighs the storage overhead for the backlinks.

Figure 3.11 shows the memory usage over minPts (ϵ = 3, log-log scale). The memory consumption of Sym-Clust and Join-Clust is stable w.r.t. increasing values of minPts, while the memory usage of Spread slightly increases. This is due to the number of concurrently stored backlinks: the larger minPts, the higher the chance that a succeeding core neighbor is not yet classified, which triggers the creation of a backlink entry. The memory grows slowly with increasing minPts and does not limit the scalability of Spread. We include the results for BMS-POS and KOSARAK, ϵ = 3; other datasets and ϵ values show similar results.

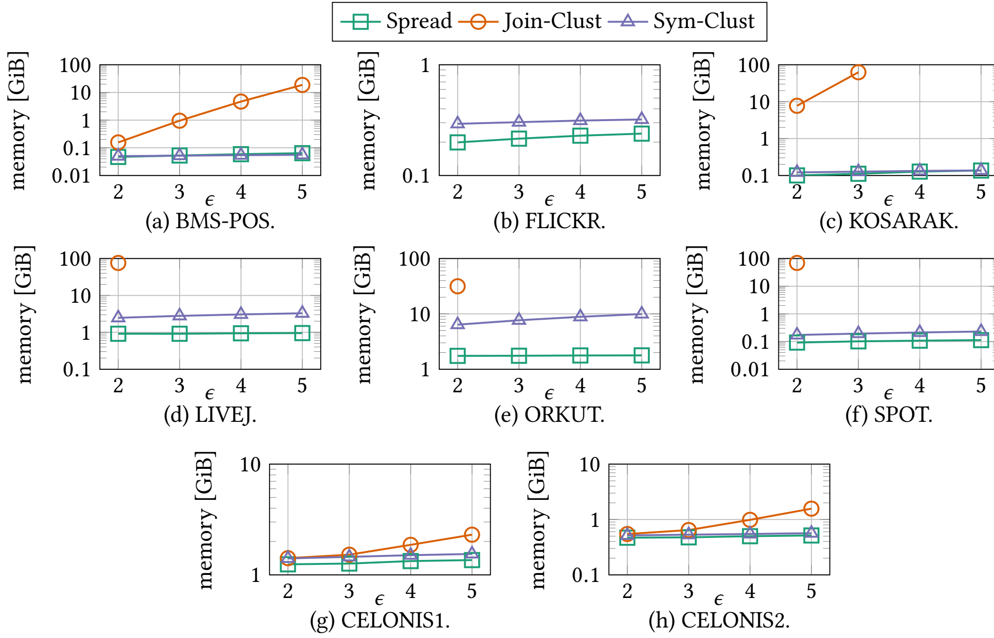


Figure 3.10: Main memory over ϵ, minPts = 16.

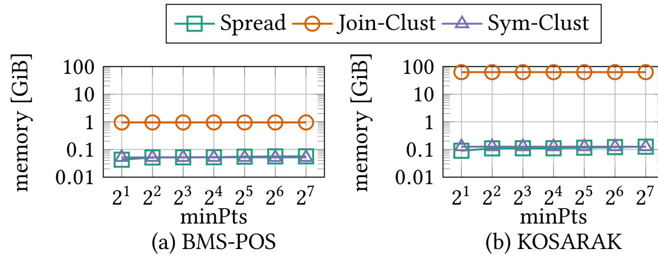


Figure 3.11: Main memory over minPts, ϵ = 3.

BACKLINKS PEAK We evaluate the effect of releasing the backlinks of a set in Spread after the set has been processed (cf. line 34, Algorithm 13). Figures 3.13 and 3.14 show the peak number of allocated backlinks relative to the maximum number of backlinks for varying ϵ (minPts = 16) and minPts (ϵ = 3), respectively. Since two backlink structures, core (green) and non-core (orange), are maintained for each set in R, at

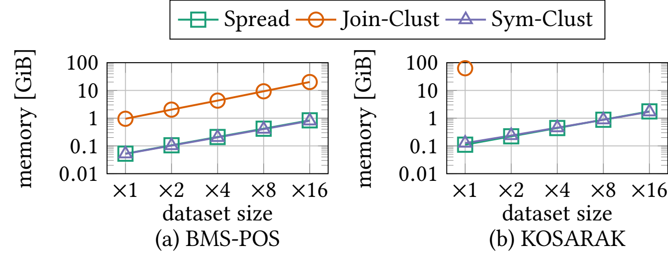


Figure 3.12: Main memory over data size, $\epsilon = 3$, minPts = 16.

most $2|R|$ backlinks can be allocated (light blue). Deallocating the backlinks of probed sets is highly effective: Only a small fraction of the maximum number of backlinks is allocated at any point in time. For increasing values of ϵ and minPts also the number of allocated backlinks grows.

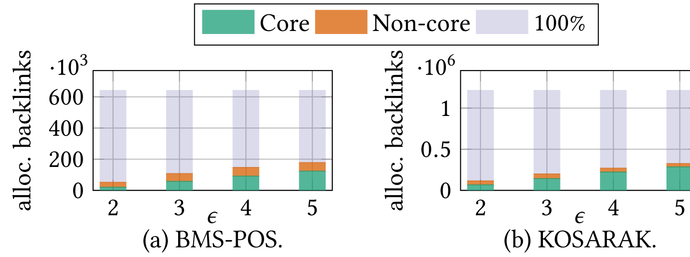


Figure 3.13: Backlinks peak over ϵ, minPts = 16.

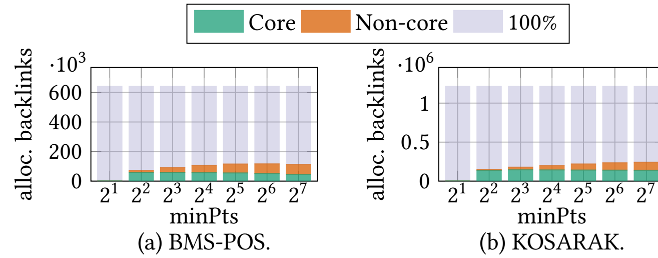


Figure 3.14: Backlinks peak over minPts, $\epsilon = 3$.

3.5.4 *Scalability*

We evaluate the scalability of Spread and its competitors to increasing data sizes. To this end, we increase the size of BMS-POS and KOSARAK using the procedure of Vernica et al. [118]. This approach does not affect the token universe, and the number of similar pairs in the dataset increases linearly with the data size.

Figure 3.9 (runtime) and Figure 3.12 (main memory) report the results for our default parameter setting. Spread shows runtimes similar to Join-Clust and outperforms Sym-Clust by a factor of about 12 (BMS-POS) resp. 5.7 (KOSARAK) on the largest dataset

($\times$16). As we increase BMS-POS by a factor of 16, the runtime increases by a factor of 195 for Spread, 204 for Join-Clust, and 460 for SymClust. The memory grows linearly for all measured data points and increases by a factor of about 2 when we double the data size. Join-Clust requires 18-25 (BMS-POS) resp. 499-569 (KOSARAK) times more memory than its competitors and runs out of memory on KOSARAK except for the $\times$1 dataset. Summarizing, Spread clearly outperforms Sym-Clust in runtime (by a factor of 5-12) and Join-Clust in memory usage (by more than an order of magnitude) as we increase the data size.

3.6 RELATED WORK

INDEXES FOR SETS Most set similarity joins operate on an inverted list index that maps signatures to candidate sets. Various signatures have been proposed [16, 29, 108, 123]. Prefixes [29] in conjunction with the length filter [6] have been shown to prune sets effectively. More sophisticated filters include positional and suffix filter [130], the removal filter [98], the position-enhanced length filter [80], and the adaptive prefix filter [122]. Wang et al. [124] leverage the similarity of the sets in an ϵ-neighborhood to reduce the overall number of false positives. Dong et al. [36] propose a size-aware algorithm that runs in $o(n^2) + O(k)$ time for k result pairs. Qin and Xiao [96] propose the pigeonring, a generalization of the pigeonhole principle that yields stronger constraints. Indexing and join techniques for sets have been studied extensively in the single-machine [81] and the distributed context [45].

Most of these approaches focus on self-joins, which order the sets and compute the lookahead neighborhood to avoid symmetric distance computations. In our work, we use the prefix filter, but any of the other asymmetric indexes is applicable.

EFFICIENT REGION QUERIES Ester et al. [40] propose the first exact DBSCAN algorithm with $O(n \log n)$ runtime for vectors of arbitrary dimension. $O(n \log n)$ runtime holds for a small number of neighbors (compared to n) and an index with $O(\log n)$ lookup time. Henceforth, efficient region query computation has been of great interest and many improvements have been proposed. Brecheisen et al. [25] use minPts-nearest neighbor queries to identify core points and postpone the other distance computations until the distances are required to get a correct DBSCAN clustering. The proposed *Xseedlist* data structure is designed for expensive distance functions and assumes a cheap but selective filter. These assumptions do not hold for sets: The verification (i.e., distance computation) of candidate pairs has shown to be highly efficient [81] (a small number of integer comparisons). Brecheisen et al. must insert the candidates into the Xseedlist data structure, which maintains sorted lists of candidates. Due to the expensive sorting procedure, we do not expect Xseedlist to improve the DBSCAN algorithm for sets. TI-DBSCAN [71] exploits the triangle inequality to reduce the search space of region queries. The solution is not index-based, sorts the points w.r.t. a reference point, and shifts a window of size 2ϵ over the sorted points. The reference point is the point with minimal values in all dimensions. This is equivalent to the empty set, and our processing order in combination with the prefix index for

sets subsumes this technique. Patwary et al. [90] introduce PARDICLE, a parallel approximate density-based clustering algorithm for Euclidean space. Its aim is to reduce the neighborhood computation time by sampling high-density regions. Kumar and Reddy [72] propose a new graph-based index structure called Groups. It discovers groups of patterns in two scans over the dataset and applies a standard DBSCAN afterwards. Groups accelerates region queries by pruning noise points effectively. This technique assumes Euclidean distance and does not consider Hamming distance or other set similarity measures. Recently, Jiang et al. [64] proposed SNG-DBSCAN, which prevents the computation of the full ϵ-neighborhood graph via subsampling its edges. This results in $O\left(sn^2\right)$-time complexity with s being the sampling rate. Under certain distribution assumptions, SNG-DBSCAN has been shown to preserve the exact ϵ-neighborhood graph for $s \approx (\log n)/n$ with $O(n \log n)$ runtime.

DBSCAN TECHNIQUES. Yang et al. [132] propose the distributed DBSCAN-MS clustering algorithm for metric spaces. DBSCAN-MS uses pivots to map the data from metric space to vector space, where it is partitioned in order to be distributed. A local DBSCAN is then executed on each partition. Our solution does not rely on the metric properties of set distances, but uses specialized set indexes. However, our techniques may be leveraged in the context of DBSCAN-MS, where the data points are ordered by one of the dimensions for efficient neighborhood queries.

Patwary et al. [89] propose PDSDBSCAN, a parallel DBSCAN algorithm that uses the disjoint-set data structure to connect data points into clusters. We only insert links between subclusters into the disjoint-sets structure, while PDSDBSCAN inserts a link for each neighbor, rendering the number of required union operations a bottleneck for this approach.

Böhm et al. [20] use a block-nested loop join and buffer the join result to reduce the number of block accesses required to compute ϵ-neighborhoods. CUDA-DClust [21] is a GPU-based solution that splits clusters into chains that are expanded from different starting points in parallel. In order to merge chains into clusters, a quadratic-size bit matrix is used. We maintain only a linear number of links and leverage disjoint-sets to merge clusters. Incremental DBSCAN algorithms [41] deal with updates on an existing clustering. Similar to our approach, these techniques may need to merge clusters when new points are inserted. None of the above solutions supports asymmetric neighborhood indexes.

Numerous parallel and distributed algorithms [32, 34, 50, 53–55, 63, 99, 104, 126, 131] as well as approximations [48, 79, 120, 128] have been proposed. We present an exact, single-core solution for sets.

3.7 CONCLUSION

In this chapter, we have investigated clustering techniques for large collections of sets. Our work was motivated by an application in process mining that models processes as sets to assess their similarity. We have shown that the solutions that are currently available, Sym-Clust and Join-Clust, are not satisfying: Sym-Clust is slow since it

cannot use effective asymmetric set indexes, while Join-Clust is infeasible for many settings due to its excessive memory usage. We introduced a novel, density-based clustering algorithm, Spread, that can process data points in any user-defined order and is therefore fit for the use with asymmetric indexes. Spread combines the best of both worlds: It uses the effective asymmetric index of Join-Clust, but like Sym-Clust does not need to materialize the neighborhoods. We introduced so-called backlinks to guarantee a correct DBSCAN clustering and showed the correctness of our approach. To the best of our knowledge, Spread is the first DBSCAN-compliant algorithm that uses an asymmetric index and runs in linear space.

Spread uses the index as a black box and works with any data type. Interesting future work includes evaluating the performance of Spread for vector data, where candidates are generated using a sliding window that is shifted along one dimension. The data points in the window are candidates, i.e., the window simulates an asymmetric index for Spread.

ACKNOWLEDGMENTS We thank Alexander Miller, Mateusz Pawlik, Thomas Hütter, Manuel Widmoser, Manuel Kocher, Daniel Ulrich Schmitt, Konstantin Thiel, Daniel Grittner, Christian Böhm, and Claudia Plant for valuable discussions, and Manuel Kocher for typesetting Figures 3.1 and 3.3. This work was partially supported by the Austrian Science Fund (FWF): P 29859.

4 A MULTI-CORE SOLUTION FOR DENSITY-BASED CLUSTERING OF SETS

The Spread algorithm (cf. Chapter 3) is designed for single-threaded execution. In practice, however, parallel algorithms are desirable to exploit the full potential of modern multi-core processors. Therefore, we introduce MC-Spread, an extension of Spread to multi-core environments.

We summarize the contributions of this chapter as follows.

- We propose MC-Spread, a multi-core extension for the sequential Spread algorithm. For a total number of $k + 1$ threads, MC-Spread splits the threads into k threads that compute (distinct) lookahead neighbors and one thread that builds the clusters. Each neighborhood thread stores its neighborhoods into a shared array and notifies the clustering thread about the availability of the respective neighborhood. The clustering thread processes (and frees) the lookahead neighborhoods concurrently in a similar manner to Spread, i.e., using disjoint-set data structure and backlinks (cf. Chapter 3).

- The neighborhood threads may fill up the memory if they allocate neighborhoods much faster than the clustering thread frees them. To prevent this, we propose a memory constraint that bounds the memory the neighborhood threads are allowed to occupy.

- We empirically evaluate MC-Spread against MC-Join-Clust, a multi-core extension of the materialization-based solution Join-Clust (cf. Chapter 3). We observe that MC-Spread scales better with the number of cores in terms of runtime. Furthermore, we discuss characteristics of datasets and configurations that affect the scalability of both approaches. Furthermore, we discuss experimental results of the memory-constrained version of MC-Spread.

The remainder of this chapter is organized as follows. In Section 4.1, we cover some preliminaries on multi-core processors and concurrency. Section 4.2 presents Simple-MC-Spread, a simple multi-core extension of the Spread algorithm. In Section 4.3, we discuss some effects that limit the scalability of Simple-MC-Spread and propose refinements to mitigate them. Our multi-threaded implementations of Spread and Join-Clust are presented in Section 4.4 (MC-Spread) and Section 4.5 (MC-Join-Clust), respectively. In Section 4.6, we evaluate the multi-threaded solutions MC-Spread and MC-Join-Clust against each other with respect to runtime, memory consumption, and caching. We conclude this chapter with an outlook to future work in Section 4.7.

4.1 PRELIMINARIES

Before we discuss our multi-core extension, we give a brief introduction to multi-core processors and discuss some phenomena that affect the performance of concurrent code and will be relevant for the discussion later in this chapter (e.g., false sharing).

MULTI-CORE PROCESSOR BASICS A *multiprocessor system* is a system with several processors (CPUs). Typically, each processor has its own memory and resides on a separate chip. In a *multi-core* processor, multiple processors (often referred to as *cores*) are physically located on the same chip with shared memory [101]. The memory hierarchy of modern multi-core systems consists of multiple levels (with the fastest memory type on top). We consider a simplified memory hierarchy that covers the basic memory types: *multi-level caches* (L1, L2, and L3 cache) and *main memory* [62]. *Main memory* is always shared across all cores over all processors and provides the largest storage. *Caches* are physically closer to the cores (and thus faster) but have lower storage capacities [62]. The *L1* and *L2 caches* are the fastest cache levels (in this order) and each core often has an isolated, private L1 and L2 cache. Contrarily, the *L3 cache* is typically larger but shared across all cores. The main purpose of caches is to speed up the access to a particular memory location by reducing the latency [101]. Without caches, an access to a memory location has to be served by main memory (which has a high latency because it is physically far away from the CPU). When a memory location is accessed, the CPU checks the first level of the memory hierarchy (i.e., the L1 cache, in our model). If the first level does not contain the data, the CPU checks the next level (i.e., the L2 cache). In the worst case, the CPU has to load the data from main memory. In this case, all cache levels (L1, L2, and L3) are updated such that the next memory access is hopefully served from one of the caches (ideally the L1 cache) [62]. Different locality principles (also referred to as *locality of reference*) are used to maintain the content of a cache such that it favors low latency [62]. Two popular locality principles are *spatial* and *temporal locality*. *Spatial locality* is based on the observation that an access to a particular memory location is often followed by an access to nearby memory locations. In contrast, *temporal locality* refers to the fact that the same memory location is often accessed repeatedly [62]. A cache is typically organized in so-called *cache lines*, which are blocks of fixed size (e.g., 64 bytes). A cache line is the smallest data unit that is loaded from memory into cache, i.e., not only a single memory location is loaded but also nearby locations such that they fill the cache line [62].

The cache hierarchy stores redundant copies of the data. In the case of a non-shared cache, e.g., L1 or L2, a cache line may be invalidated by another core that updates the data in its private cache. Consequently, *cache coherence* protocols have been developed to keep the cache lines coherent [62]. For example, many processors implement a cache coherence protocol that is based on *invalidation*, i.e., if one core modifies a cache line, then the corresponding cache lines of other cores are invalidated ("dirty") and must be updated before the next access [7, 107]. Keeping the cache coherent does not come for free and incurs overhead.

Finally, common terms in the context of caches are *cache hit* and *cache miss*. A *cache hit* occurs if the data required by an instruction is present in the cache, otherwise the data must be loaded from memory (called *cache miss*) [62]. An invalidation-based cache coherence protocol may also cause cache misses if the required memory location is in the cache, but the cached copy is invalid. The performance of a multi-threaded application often depends on the ratio of cache hits and misses [127].

EFFECTS IN CONCURRENT CODE Concurrent code adds complexity compared to its sequential counterpart, and a number of side effects that impact the performance may occur. Ideally, concurrent code runs x times faster than its sequential counterpart if it is executed on x cores instead of only a single one (linear scalability) [127]. In practice, however, concurrent code often does not scale linearly due to several effects. In the worst case, the performance of concurrent code may even degrade compared to its sequential counterpart. In the following, we cover some basic effects that degrade the performance of concurrent code in practice [127]. We assume a multi-core processor system with one thread being executed on a dedicated core.

We already mentioned the cache coherence problem. The fact that caches have to be kept coherent also affects the performance of a multi-threaded execution if the data is subject to modifications. In the case of read-only accesses to the data, the data copies in the cache lines are coherent by definition. But if one thread modifies the data that is held in the cache, the cache coherence protocol triggers an update for the caches of the other cores. In other words, the other cores have to wait until the cache is updated [127]. Recurring, mutual updates of the cache lines is often referred to as *cache ping-pong* and may impact the performance of concurrent code significantly when the number of threads increases [127].

Another phenomenon related to caches is *false sharing*, which is caused by the fact that caches load data at the granularity of cache lines and cache coherence protocols also work at that granularity [127]. Assume that two threads T_1 and T_2 access two different memory locations x_1 resp. x_2 that are physically close in main memory such that they reside on the same cache line. If T_1 updates x_1 before T_2 tries to read x_2, the cache line of T_2 is invalidated and must be updated. The two threads do not share data, but the caches *falsely share* the cache lines, which is one possible reason for cache ping-pong [127].

Finally, we briefly discuss *contention* on data due to atomic operations. In sequential code, the instructions are mostly executed in order (except for, e.g., instruction reordering done by the compiler or the CPU) [127]. In a multi-threaded application, the instructions of an operation of thread T_1 may interleave with the instructions of an operation of thread T_2 (due to the scheduling algorithm of the operating system). As a result, a shared memory location may be subject to a *race condition* [127]: Two (or more) threads try to update the data of a shared memory location at the same time. Atomic operations are one mechanism to avoid undefined behavior due to race conditions. An atomic operation is a single, indivisible operation, which often reads *and* writes a value simultaneously (commonly referred to as *read-modify-write*) [127]. For example, incrementing a variable x can be done atomically. If a shared counter

is incremented atomically by two threads, T_1 and T_2, then T_2 may have to wait until T_1 successfully executed the read-modify-write operation (because T_2 must read the updated counter value). If many threads wait for each other, then the contention is high and the application may progress slowly [127].

4.2 A SIMPLE MULTI-CORE EXTENSION OF SPREAD[1]

The Spread clustering algorithm (cf. Chapter 3) is designed for single-core execution. All line numbers referenced in this section refer to Algorithm 13 in Section 3.4.3. We sketch an extension to multi-core processors that requires little synchronization between threads. Our extension is based on the observation that Spread spends most of the runtime in neighborhood computations (lines 6-11; cf. Algorithm 13). While for some datasets the neighborhood computation accounts for only about half of the overall runtime (e.g., 55% for ORKUT, $\epsilon = 3$), for the configuration with the highest runtime in our experiments (CELONIS1, $\epsilon = 5$), Spread spends over 99% of the runtime in computing the neighborhoods.

We distribute the workload to $k + 1$ threads, $T_1, T_2, \ldots, T_{k+1}$. Threads T_1-T_k are responsible for the neighborhood computations (lines 6-11; cf. Algorithm 13), T_{k+1} performs the actual clustering (lines 12-34; cf. Algorithm 13). The runtime of the other steps in Algorithm 13 (cf. Section 3.4.3) is negligible.

Neighborhood Computation. Let $r_i \in R$, $1 \leq i \leq |R|$ be the i-th set of R in processing order. Thread T_j, $1 \leq j \leq k$, computes the neighborhoods $N_\epsilon^\succ(r_i)$ of all r_i with $j = i \bmod k$ (i.e., round robin). Each thread processes the assigned sets r_i in processing order (i.e., increasing values of i). The neighborhood computation in Algorithm 13 is interleaved with updating the density counters of r_i and its neighbors. Only this step requires synchronization (e.g., using atomic writes) since multiple threads may access the same counter concurrently. We do not expect congestions since the density updates are distributed over all neighbors.

Cluster Scan. Thread T_{k+1} scans the sets in processing order and performs the steps in lines 12-34 (maintain backlinks and disjoint-set, assign preliminary cluster IDs; cf. Algorithm 13). After processing a set r_i, the memory for the neighbors of r_i is released.

Synchronization. We need to make sure that T_{k+1} processes set r_i only after r_i's neighbors have been computed. This can be achieved with a lock (implemented as condition variable[2]) on r_i that is held by T_j, $j \leq k$, until the neighborhood of r_i is computed. T_{k+1} needs to get the lock on r_i before processing it.

Memory. T_{k+1} releases the neighbors after processing them. If the parallel neighborhood computation is faster than T_{k+1}, the precomputed neighborhoods will fill up the memory. This is avoided with a shared counter that is incremented by T_1-T_k (when they process a new set r_i) and is decremented by T_{k+1} (after processing r_i). The neighborhood computation of r_i is postponed until the counter is below some threshold that bounds the number of concurrently materialized lookahead neighborhoods.

1 The concept described in this section was published in Kocher et al. [70] (EDBT 2021).
2 A queue of threads waiting for a condition to become true.

4.3 REFINING THE SIMPLE ALGORITHM

We call the multi-core extension of Spread discussed in Section 4.2 the *Simple-MC-Spread* algorithm. Simple-MC-Spread assumes a total number of $k + 1$ threads and splits them into two types of threads: (i) *Neighborhood threads* with the primary task of computing the lookahead neighborhood $N_{\epsilon}^{\succ}(r)$ for a particular probing set r (lines 6-11; cf. Algorithm 13). Moreover, a neighborhood thread updates the density counters atomically once the respective lookahead neighbors are known. (ii) A *clustering thread* with the primary task of assigning the cluster identifiers (lines 12-34; cf. Algorithm 13). This includes the (non-concurrent) maintenance of the disjoint-set data structure and the backlinks, and assigning the final cluster IDs. Based on our observation that the neighborhood computation consumes most of the runtime for some configurations (e.g., over 99% for CELONIS1 and $\epsilon = 5$), Simple-MC-Spread spawns k neighborhood threads, T_1-T_k, and a single clustering thread, T_{k+1}. The sets are assigned in a round-robin fashion. Simple-MC-Spread was implemented in C++ (2017 standard). Subsequently, we discuss pitfalls of the Simple-MC-Spread algorithm and propose solutions.

4.3.1 *Idle Clustering Thread*

We observe that the neighborhood threads may be too slow. Simple-MC-Spread assigns each neighborhood thread T_1-T_k a fixed number of sets in a round-robin fashion. We pre-allocate a shared array of size $|R|$ that stores the lookahead neighborhoods and maintain a condition variable cv_i for each array entry $1 \leq i \leq |R|$. After computing the lookahead neighbors of a probing set r_i, the associated thread (i) increments the density counters, (ii) moves the lookahead neighbors to the neighborhoods array, and (iii) unlocks cv_i of the corresponding entry in the array and notifies the clustering thread. However, for some configurations and datasets (e.g., high ϵ values; CELONIS1 and CELONIS2), we observe that the clustering thread has to wait for the neighborhood threads to unlock the lookahead neighborhoods. We attribute this to the expensive neighborhood computation in case of the CELONIS datasets (i.e., the neighborhoods are small but an excessive number of candidates must be verified, cf. Table 3.3). Therefore, we adapt the clustering thread such that it helps with the neighborhood computation if required. This prevents the clustering thread from idling while waiting for a neighborhood thread to notify it. To this end, we introduce a shared counter, *next_id*, that represents the ID of the next set to be probed against the index. Each thread atomically fetches and decrements *next_id* to get the next ID it has to probe (instead of assigning the IDs in a round robin fashion). When the clustering thread is supposed to cluster a particular set r_i, it (atomically) checks if the corresponding neighborhood is available. If it is available, then r_i is clustered. Otherwise, the clustering thread helps computing the neighborhoods, i.e., it atomically fetches and decrements *next_id*, and computes the corresponding lookahead neighborhood (in the manner of a neighborhood thread). This is done until the neighborhood of r_i is available. We expect this approach to prevent the idle clustering thread and provide better load balancing (as the cost of a neighborhood computation is unknown). We did not observe contention on *next_id*.

4.3.2 *Cache Misses and False Sharing*

We observe that incrementing the shared density counters in the neighborhood threads may not be cache-friendly. Simple-MC-Spread increments the shared density counters in the neighborhood threads. We observe that this is problematic for configurations with large neighborhoods, e.g., the configurations for which Join-Clust runs out of memory in Section 3.5 (with the FLICKR dataset being the most extreme case). Many of these datasets contain a large portion of small sets that all are pairwise neighbors. For example, any two sets of size 2 are similar under a Hamming distance of $\epsilon = 4$. For datasets like KOSARAK and FLICKR, we observe that Simple-MC-Spread does not scale well with the number of cores. In the case of the FLICKR dataset, the performance even degrades compared to our single-threaded implementation of Spread. After profiling the executions with Linux `perf`[3] tools, we observe that the number of CPU cycles grows with the number of threads, whereas the number of instructions is constant. Figures 4.1 and 4.2 show the wallclock time and the number of CPU cycles for CELONIS1, KOSARAK, and FLICKR[4] ($\epsilon = 3$; Simple-MC-Spread and MC-Join-Clust, cf. Section 4.5), respectively. In Figure 4.1, the dashed lines show the respective single-threaded runtimes if scaled linearly with the number of cores, i.e., the single-threaded runtime is divided by the number of cores. For the FLICKR dataset, the number of CPU cycles for 16 cores is almost 30 times higher compared to the single-threaded execution, although both algorithms execute about $1.6 \cdot 10^{12}$ instructions.

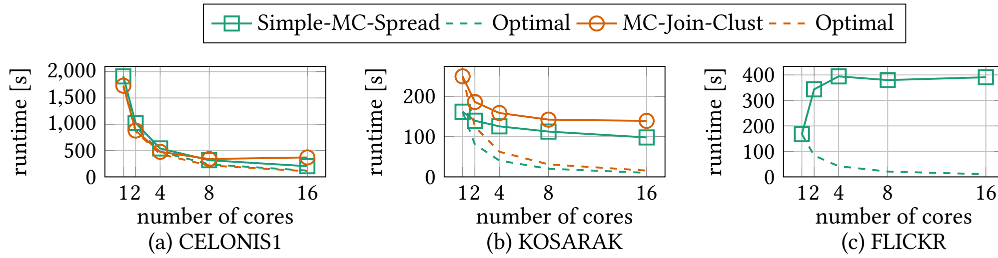


Figure 4.1: Wallclock time for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16.

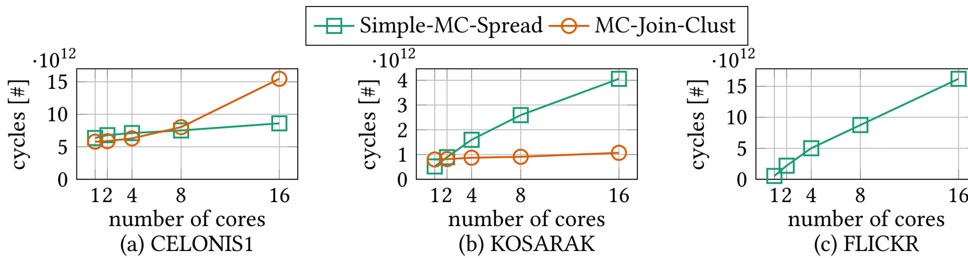


Figure 4.2: CPU cycles for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16.

3 https://man7.org/linux/man-pages/man1/perf.1.html
4 Note that MC-Join-Clust runs out of memory for $\epsilon = 3$, thus no data points are included for FLICKR.

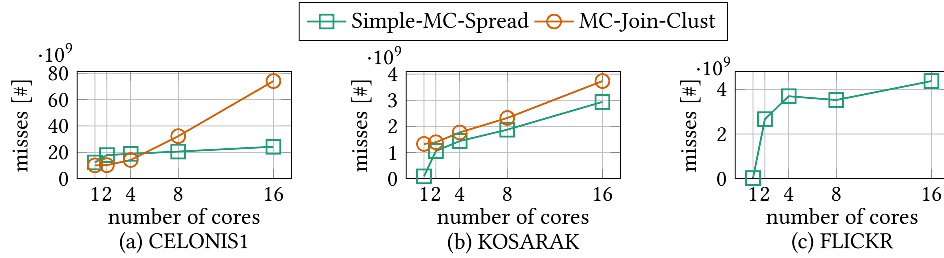


Figure 4.3: Cache misses for CELONIS1, KOSARAK, and FLICKR, $\epsilon = 3$, minPts = 16.

Further profiling revealed that also the number of cache misses grows rapidly, cf. Figure 4.3. In the case of CELONIS1, a similar effect can be observed for the multi-core implementation of Join-Clust, MC-Join-Clust (cf. Section 4.5). However, we note that the CPU cycles and cache misses of MC-Join-Clust increase at a much lower rate (i.e., about 3 times more CPU cycles for 16 cores compared to the single-threaded execution). For Simple-MC-Spread, we conclude that there are two reasons for this effect: (i) Contention on the atomic density counters (many different threads may access the same counters concurrently). This was confirmed by a test in which we remove the atomicity from the density counters. Despite reporting incorrect results, we observe a much lower CPU cycle growth rate and better performance. (ii) False sharing of the density counters. Different threads updating the counters may lead to alternating invalidation of the cache lines (one counter occupies 4 bytes, and a typical cache line size is about 64 bytes). To mitigate these effects, we move the update procedure of the density counters into the clustering thread. Although this change increases the load on the sequential part of Simple-MC-Spread, the additional load is low and leads to better CPU utilization and runtime. Improving over this adaptation may be subject to further investigations.

4.3.3 *Controlling the Memory*

Apart from these adaptations, we note that the reported results are without a mechanism to bound the number of lookahead neighborhoods that are materialized simultaneously in main memory (as described at the end of Section 4.2). In our experiments, we observe that this results in running out of memory for some configurations (cf. Section 4.6). Thus, we also test a memory-constrained version of MC-Spread, which maintains an atomic counter, *alloc_mem*, that represents the allocated memory. We approximate the memory consumption by summing up the allocated memory for the lookahead neighborhoods (including some overhead). The memory constraint, *mem_bound*, is given as command-line argument and represents the amount of memory (in GiB) that the algorithm is allowed to occupy.

Initially, *alloc_mem* is set to zero. Before computing a lookahead neighborhood, a neighborhood thread (atomically) checks if the memory constraint is satisfied, i.e., if $alloc_mem \leq mem_bound$. If this is not the case, the neighborhood thread spins until the constraint is satisfied. For each computed lookahead neighborhood, the

corresponding thread (atomically) increases *alloc_mem* by the memory that is allocated for the corresponding neighborhood. After processing a lookahead neighborhood, the clustering thread decreases *alloc_mem* by the space that is freed. Although we only approximate the allocated memory, this mechanism effectively bounds the memory that is occupied by the neighborhoods. Consequently, we are able to execute all configurations in our experiments if the memory constraint is enabled (cf. Section 4.6).

4.4 MULTI-CORE SPREAD

Previous sections already describe the principal approach for Spread's multi-core extension, denoted *MC-Spread*. In this section, we implement MC-Spread *without* memory constraint (to simplify the discussion). Before we give the corresponding pseudocode, we recap the most important data structures.

Given $k+1$ threads, we spawn k neighborhood threads, T_1-T_k, and a single clustering thread T_{k+1}. The following structures are shared between all threads: The read-only inverted index I, a lookahead neighborhood array, ln, of size $|R|$ (where $ln[i]$ stores the lookahead neighbor of the i-th set, r_i), and a shared atomic counter, *next_id*, that represents the ID of the next set to be probed (i.e., compute the lookahead neighbors). We initialize *next_id* to $|R|$ (the ID of the first set in our processing order), and use the operation *next_id.fetch_decr*() to read and decrement *next_id* atomically. Furthermore, each set r is associated with a condition variable $r.cv$, which locks the corresponding entry in the neighborhood array.

For the condition variable $r.cv$ we assume the following operations: $r.cv.wait()$ causes a thread to wait for the lookahead neighborhood of r, $r.cv.notify()$ notifies all threads that wait for the neighborhood of r, and $r.cv.would_wait()$ returns true if the call to $r.cv.wait()$ would result in waiting (false otherwise). Note that if $r.cv.notify()$ is called before $r.cv.wait()$, then the calling thread does not have to wait. $r.cv.init()$ initializes $r.cv$ such that the corresponding entry in the neighborhoods array is locked. We refer to Williams [127] for a discussion on the use of condition variables in C++.

Algorithm 14: MC-Spread(R, ϵ, minPts)

Input: Collection of sets R, distance threshold ϵ, min. density minPts
Result: A correct DBSCAN clustering of R w.r.t. ϵ, minPts

```
// shared structures between all threads
1 I ← Create-Index (R, ε); // read-only inverted index
2 ln ← new associative array of size |R|; // lookahead neighborhood array
3 next_id ← new atomic counter with value |R|; // 1-based indexing
4 foreach r ∈ R do
5     r.dens ← 1; r.cid ← −∞; r.cv.init();
6 do concurrently using k threads // assuming k + 1 hardware threads in total
7     Compute-Neighborhoods(R, ε, I, ln, next_id);
8 MC-Cluster(R, ε, minPts, I, ln, next_id); // the main thread clusters concurrently
```

Algorithm 14 gives the pseudocode of MC-Spread's main procedure. A neighborhood thread (i) reads and decrements *next_id* atomically and (ii) probes the next set (the read

```
Algorithm 15: Compute-Neighborhoods(R, ϵ, I, ln, next_id)
  Input: Collection of sets R, distance threshold ϵ, inverted index I,
         neighborhood array ln, atomic counter next_id
  Result: The neighborhoods computed by this neighborhood thread
1 id ← next_id.fetch_decr(); // fetch ID of next set to probe
2 while id > 0 do
3     Probe-Notify(R, ϵ, I, ln, id); // probe set with ID id and notify clustering thread
4     id ← next_id.fetch_decr(); // fetch ID of next set to probe
```

```
Algorithm 16: Probe-Notify(R, ϵ, I, ln, id)
  Input: Collection of sets R, distance threshold ϵ, inverted index I,
         neighborhood array ln, probing ID id
  Result: Lookahead neighbors of set with ID id
1 r ← set with ID id; // get set to probing ID
2 r.cv.lock(); // lock corresponding entry in neighborhoods array (if not locked yet)
3 M ← Probe (r, I, ϵ); // get candidates for r
4 foreach (s, po) ∈ M do // po . . . prefix overlap
5     if Verify-Pair(r, s, ϵ, po) then
6         ln[r] ← ln[r] ∪ {s}; // append lookahead neighbor to shared array
7 r.cv.notify(); // neighborhood of r is ready; unlock entry and notify clustering thread
```

```
Algorithm 17: MC-Cluster(R, ϵ, minPts, I, ln, next_id)
  Input: Collection of sets R, distance threshold ϵ, min. density minPts, inverted index I,
         neighborhood array ln, atomic counter next_id
  Result: A correct DBSCAN clustering of R w.r.t. ϵ, minPts
1  ds ← new disjoint-set; nc_bl, c_bl ← new backlinks;
2  foreach r ∈ R do ds.make_set (r.id) ;
3  foreach r ∈ R in processing order do
4      while r.cv.would_wait() do // help with neighborhood computation
5          help_id ← next_id.fetch_decr(); // fetch ID of next set to probe
6          if help_id > 0 then Probe-Notify(R, ϵ, I, ln, help_id) ;
7      r.cv.wait(); // wait for the neighborhood to be ready
8      foreach s ∈ ln[r] do // update neighbor densities
9          r.dens ← r.dens + 1; s.dens ← s.dens + 1;
10     Cluster(r, minPts, ln, ds, nc_bl, c_bl); // clustering procedure like in Spread
11     release ln[r]; // release neighborhood; not needed anymore
12 foreach r ∈ R do // final assignment of cluster IDs
13     if r.cid ≠ −∞ then r.cid ← ds.find_set (r.cid);
```

value of *next_id* is the corresponding set ID) against the inverted index to retrieve its lookahead neighbors. The two steps are executed while $next_id > 0$ holds (i.e., there are still sets to be probed). Algorithm 15 shows the pseudocode of a neighborhood thread (Algorithm 16 is an auxiliary function). We reuse Algorithms 10–12 (cf. Section 3.3.2).

The clustering thread maintains the disjoint-set data structure as well as the backlinks. It iterates over all sets $r \in R$ in processing order and performs the following steps for

Algorithm 18: Cluster(r, minPts, ln, ds, nc_bl, c_bl)

```
Input: Current set r, min. density minPts, neighborhood array ln, disjoint-set ds,
       non-core backlinks nc_bl, core backlinks c_bl
Result: Subcluster assignments of r and its lookahead neighbors
   // Cf. also lines 12-34 in Algorithm 13
1  if r.dens ≥ minPts then // r is core
2      if r.cid = −∞ then r.cid ← r.id;
3      foreach x ∈ nc_bl[r] do // claim border sets x < r
4          if x.cid = −∞ then x.cid ← r.cid;
5      foreach s ∈ N_ε^>(r) do // s > r
6          if s.cid = −∞ then // claim unclaimed s > r
7              s.cid ← r.cid
8          else if r.cid ≠ s.cid then // s already claimed
9              if s.dens ≥ minPts then // s is core
10                 ds.union(r.cid, s.cid) // link subclusters
11             else // remember core neighbor r
12                 c_bl[s] ← c_bl[s] ∪ {r.cid}
13     foreach x ∈ c_bl[r] do ds.union(r.cid, x);
14 else // r is not core, i.e., r.dens < minPts
15     if r.cid = −∞ then // claim potential border set r
16         foreach s ∈ N_ε^>(r) do
17             if s.dens ≥ minPts then // s is core
18                 if s.cid = −∞ then s.cid ← s.id;
19                 r.cid ← s.cid; break;
20     if r.cid = −∞ then // remember potential border set r
21         foreach s ∈ N_ε^>(r) do nc_bl[s] ← nc_bl[s] ∪ {r} ;
22 release c_bl[r] and nc_bl[r] // not needed anymore
```

each single probing set r: (i) It atomically checks if the neighborhood of the current set r is available. If not, then the clustering thread also computes neighborhoods until it becomes available. (ii) The clustering thread tries to acquire a lock on the condition variable $r.cv$. This is the synchronization point between clustering thread and the neighborhood threads. (iii) The clustering thread iterates over the lookahead neighbors of r and increments the density counters accordingly. (iv) The clustering thread clusters the current set r and releases the corresponding lookahead neighbhorhood. (v) Finally, the clustering thread assigns the final cluster IDs once all sets have been processed. Algorithm 17 presents the pseudocode of the clustering thread (Algorithm 18 is an auxiliary function).

Remarks & Details. We use the standard C++ library to implement MC-Spread. Atomic counters are implemented using `std::atomic`[5]. Its `fetch_sub` function is used to read and decrement a counter atomically. Each condition variable is implemented

5 https://en.cppreference.com/w/cpp/atomic/atomic

as a triple of `std::condition_variable`[6], `std::mutex`[7], and a boolean field. The boolean field is used in the call to *wait*() to prevent spurious and lost wakeups of threads (cf. `std::condition_variable::wait`[8]) [127]. Recall that each entry in the shared lookahead neighborhood array ln has a separate condition variable triple. The k neighborhood threads and the clustering thread are executed concurrently. We spawn k neighborhood threads (using `std::thread`[9]) and the main thread continues with the clustering procedure. The condition variables ensure that the clustering thread processes only neighborhoods that are available.

4.5 MULTI-CORE JOIN-CLUST

The parallelization of the join-based baseline solution, Join-Clust, operates in two decoupled phases: (1) The join and neighborhood materialization and (2) the clustering (DBSCAN). In phase (1), we focus on the parallelization of the set similarity join and, in particular, the probing procedure since the build time of the prefix-based inverted index is negligible. The multi-core version of Join-Clust, *MC-Join-Clust*, spawns only neighborhood threads ($k + 1$ in total) and uses a sequential DBSCAN[10] (due to the low runtime once the neighborhoods are known). Each neighborhood thread T_i maintains an individual array of result pairs, $pairs_i$, which together yield the overall join result, $pairs = \bigcup_{i=1}^{k+1} pairs_i$. Since the complete ϵ-neighborhoods are materialized, we skip the merging and directly populate the neighborhoods. This is done in a sequential loop over the thread-local join results, which serves as synchronization point before the (sequential) DBSCAN algorithm is executed. The inverted index, I, is a shared, read-only data structure. Initially, the range of probing IDs is split into $k + 1$ subranges, each of which is assigned to a separate neighborhood thread. Work stealing between the neighborhood threads is used to achieve good load balancing, i.e., a subrange is further divided (and distributed) in case of an idle thread. Algorithm 19 depicts the pseudocode of the parallelized neighborhood materialization (Algorithm 20 is an auxiliary function). Again, we reuse Algorithms 10–12 (cf. Section 3.3.2).

Remarks & Details. We use Intel's Threading Building Blocks[11] (TBB) library to split the collection of sets R and to parallelize the probing using `tbb::parallel_for`[12]. Automatic splitting into subranges and work stealing between threads [121] is implemented using `tbb::blocked_range`[13] combined with `tbb::auto_partitioner`[14].

---

6 https://en.cppreference.com/w/cpp/thread/condition_variable
7 https://en.cppreference.com/w/cpp/thread/mutex
8 https://en.cppreference.com/w/cpp/thread/condition_variable/wait
9 https://en.cppreference.com/w/cpp/thread/thread
10 Parallelization of the DBSCAN algorithm is subject to active research [104, 126], but is not the focus of this chapter.
11 https://www.threadingbuildingblocks.org/
12 https://www.threadingbuildingblocks.org/docs/help/reference/algorithms/parallel_for_func.html
13 https://www.threadingbuildingblocks.org/docs/help/reference/algorithms/range_concept/blocked_range_cls.html
14 https://www.threadingbuildingblocks.org/docs/help/reference/algorithms/partitioners/auto_partitioner_cls.html

Algorithm 19: MC-Materialize-Neighborhoods(R, ϵ)

```
   Input: Collection of sets R, distance threshold ϵ
   Result: All neighborhoods in R w.r.t. ϵ
   // shared structures between all threads
1  I ← Create-Index (R, ϵ); // read-only inverted index
2  do in parallel using k + 1 threads
3  |  R′ ← subrange for thread i, 1 ≤ i ≤ k + 1;
4  |  Compute-Pairs(R′, ϵ, I);
5  neighborhoods ← new associative array of size |R|;
6  foreach thread t_i, 1 ≤ i ≤ k + 1 do
7  |  foreach (r, s) ∈ pairs_i do // pairs stored with thread i
8  |  |  neighborhoods[r] ← neighborhoods[r] ∪ {s};
9  |  |  neighborhoods[s] ← neighborhoods[s] ∪ {r};
10 return neighborhoods
```

Algorithm 20: Compute-Pairs(R', ϵ, I)

```
   Input: Subrange of sets R′, distance threshold ϵ, inverted index I
   Result: Set of similar pairs in R′ w.r.t. ϵ
   // thread-local data structures of thread i, 1 ≤ i ≤ k + 1
1  pairs_i ← ∅;
2  foreach r ∈ R′ in processing order do
3  |  M ← Probe(r, I, ϵ);
4  |  foreach (s, po) ∈ M do
5  |  |  if Verify-Pair(r, s, ϵ, po) then
6  |  |  |  pairs_i ← pairs_i ∪ {(r, s)};
7  return pairs_i // pairs also remain in thread-local storage
```

Thread-local structures are maintained in a dedicated class that is used in combination with `tbb::enumerable_thread_specific`[15]. For our experiments over the number of cores, the maximum number of threads is set using `tbb::task_arena`[16]. The parallel for-loop is the only explicit point of synchronization.

4.6 EXPERIMENTAL RESULTS

In this section, we present experimental results for the multi-threaded extensions of the join-based baseline, MC-Join-Clust, and our algorithm, MC-Spread.

ALGORITHMS We compare the multi-threaded version of our solution, MC-Spread (cf. Section 4.4), against the multi-threaded version of the join-based approach, MC-Join-Clust (cf. Section 4.5). If not otherwise specified, MC-Spread is executed *without* memory constraint. Both algorithms are implemented in C++ (2017 standard) and follow

15 https://www.threadingbuildingblocks.org/docs/help/reference/thread_local_storage/enumerable_thread_specific_cls.html

16 https://www.threadingbuildingblocks.org/docs/help/reference/task_scheduler/task_arena_cls.html

the single-threaded implementation regarding the asymmetric prefix index, candidate generation, and efficient verification. Spread is parallelized using the standard C++ threads library and Join-Clust is parallelized using Intel's Threading Building Blocks (cf. *Remarks & Details* in Section 4.4 and 4.5, respectively).

DATASETS All experiments were executed on 8 real-world datasets: (a) Six datasets from a well-known set similarity join benchmark [45, 81]: BMS-POS, FLICKR, KOSARAK, LIVEJ, ORKUT, and SPOT. Mann et al. [81] discuss the sources of the datasets and how they are transformed into collections of sets[17]. We also test our multi-threaded algorithms on two large process mining datasets, CELONIS1-2. Each process is represented as a set (cf. Section 1.3.3). The collections of sets are deduplicated when reading the respective files from disk. Table 3.2 provides a summary of the datasets (cf. Section 3.5).

PARAMETERS Alongside the two algorithmic parameters, ϵ (neighborhood radius) and minPts (min. neighborhood density for a point to be core), our implementations take a third parameter to specify the maximum number of threads, i.e., how many physical cores are used during execution. In our experiments, we vary the number of threads and test different values of ϵ. Since the single-threaded implementations have been shown to be insensitive to minPts, we fix minPts to 16. We do not expect the multi-threaded counterparts to be sensitive to minPts because the clustering is done by a single thread (and only this thread modifies the backlinks of MC-Spread). We therefore focus on varying (i) the number of cores/threads by doubling the number of cores starting from 1, $\{1, 2, 4, \mathbf{8}, 16\}$, and (ii) the distance threshold $\epsilon \in \{2, \mathbf{3}, 4, 5\}$ (defaults in bold font).

ENVIRONMENT Experiments have been conducted on a 64-bit machine with 2 physical Intel Xeon E5-2630 v3 CPUs, 2.40GHz. Each CPU is located on a separate socket and has 8 physical cores, i.e., the total number of physical cores is 16 (hyper-threading is disabled). All sockets share 96GiB of RAM, all cores on a socket share a 20MiB L3 cache, and each core has an individual 256KiB L2 cache and 64KiB L1 cache. The machine runs Debian 10 Buster (Linux 4.19.0-12-amd64 #1 SMP Debian 4.19.152-1 (2020-10-18)). We compiled our code with `clang`[18] version 7 and highest optimization level (`-O3`). We measure the CPU time with `clock_gettime`[19] and use the elapsed time of the Linux `perf`[20] tools to measure the wallclock time (the results were similar to the results obtained with the Linux `time`[21] command). We also use the `perf` tools to measure the CPU utilization and to count the number of cache references/misses, instructions, and CPU cycles. The memory usage is the `heap peak` of Linux `memusage`[22] (using `LD_PRELOAD`). Every instance was executed in isolation (i.e., no other load on the machine).

17 http://ssjoin.dbresearch.uni-salzburg.at/datasets.html
18 https://releases.llvm.org/7.0.0/tools/clang/docs/ReleaseNotes.html
19 https://man7.org/linux/man-pages/man2/clock_gettime.2.html
20 https://man7.org/linux/man-pages/man1/perf.1.html
21 https://man7.org/linux/man-pages/man1/time.1.html
22 https://man7.org/linux/man-pages/man1/memusage.1.html

RUNTIME Figures 4.4-4.7 show our runtime results for an increasing number of cores over all values of ϵ. The runtime is the wallclock time required to partition the collection of sets into DBSCAN clusters without reading the input file. Dashed lines[23] show the single-threaded runtime divided by the number of cores, i.e., optimal linear scalability.

The missing points in the plots are due to the respective algorithm running out of memory (cf. Figures 4.8-4.11). MC-Join-Clust runs out of memory for many configurations like its single-threaded counterpart. We attribute this to the (single-threaded) materialization of growing neighborhoods. We observe that also MC-Spread runs out of memory for some datasets, $\epsilon = 5$, and a high number of threads. This is due to the fact that more lookahead neighborhoods are computed and materialized simultaneously.

We observe that both algorithms MC-Spread and MC-Join-Clust show different scaling behavior in the number of cores for different configurations. In the case of $\epsilon = 2$, the runtime of both algorithms is close to the optimal runtime when we increase the number of cores for BMS-POS, KOSARAK, CELONIS1, and CELONIS2. These configurations are characterized by rather small ϵ-neighborhoods (which is also the reason why Join-Clust does not run out of memory). Note that the computation of the neighborhoods may be expensive despite their small size, for example, in the case of the CELONIS datasets (due to many candidates). In this scenario, the clustering thread benefits the most from higher parallelism of the neighborhood threads. For some datasets, MC-Spread scales better than MC-Join-Clust with the number of cores (e.g., KOSARAK or CELONIS1).

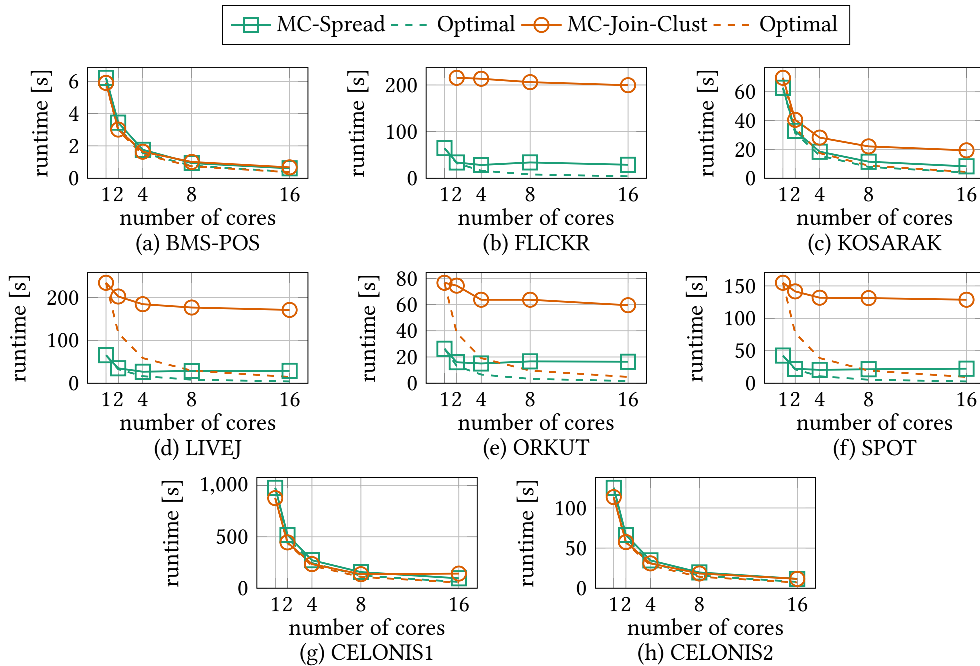


Figure 4.4: Wallclock time over the number of cores, $\epsilon = 2$, minPts = 16.

23 We do not include the dashed line for Join-Clust and the FLICKR dataset since the sequential Join-Clust runs out of memory; see discussion on memory usage, Figures 4.8-4.11.

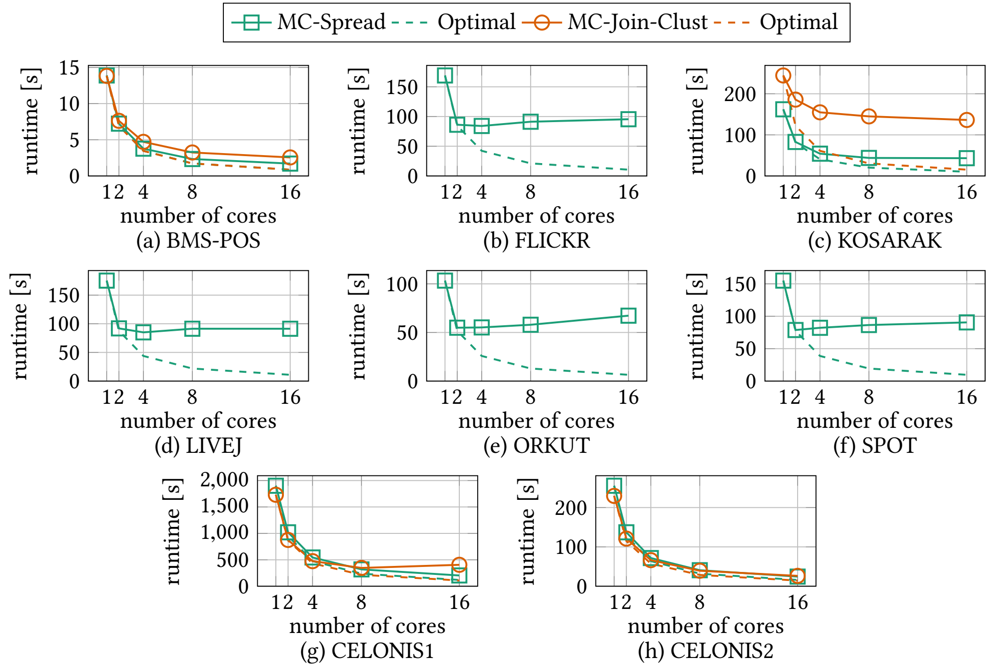


Figure 4.5: Wallclock time over the number of cores, $\epsilon = 3$, minPts = 16.

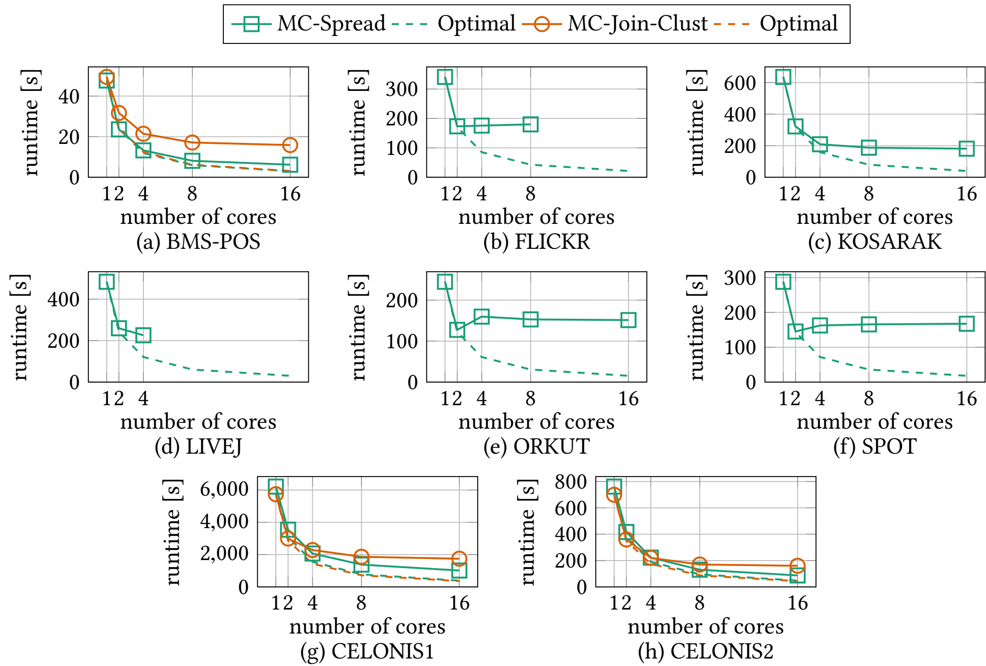


Figure 4.6: Wallclock time over the number of cores, $\epsilon = 4$, minPts = 16.

For both solutions, the other datasets (FLICKR, LIVEJ, ORKUT, and SPOT) seem to be harder due to their large neighborhoods (even for $\epsilon = 2$; cf. memory consumption of the respective instances). We observe that MC-Spread reaches a plateau for a higher

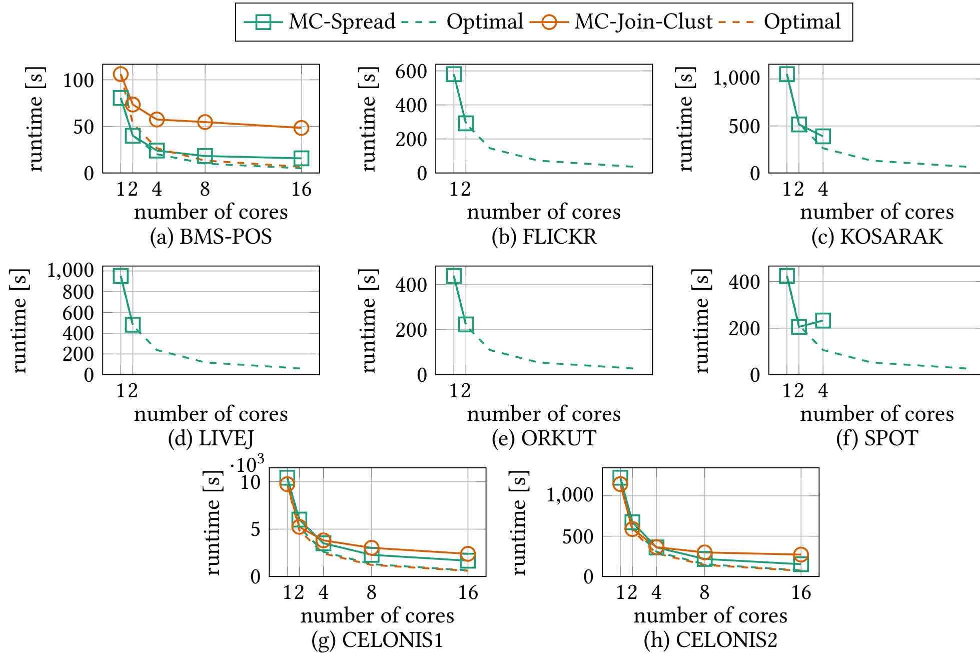


Figure 4.7: Wallclock time over the number of cores, $\epsilon = 5$, minPts = 16.

number of cores (e.g., more than 2 resp. 4 cores in case of SPOT resp. LIVEJ). This was to be expected because the clustering thread has the highest share of overall runtime for these datasets (i.e., about 41.3% for FLICKR, 34.5% for LIVEJ, 45.2% for ORKUT, and 43.7% for SPOT; $\epsilon = 3$, minPts = 16).

For higher values of ϵ, the runtime results are comparable but we make the following additional observations: (i) MC-Spread reaches a plateau also for the KOSARAK dataset and (ii) the runtime gap between MC-Join-Clust and MC-Spread increases for 16 cores and BMS-POS (due to growing neighborhoods).

We compare the speedups of MC-Spread and MC-Join-Clust in Table 4.1 for $\epsilon = 2$ (smallest neighborhoods) and $\epsilon = 5$ (largest neighborhoods). The speedup of algorithm A (for a particular ϵ and minPts) is calculated as $speedup(A) = \frac{\text{1-core time of } A}{\text{max-core time of } A}$, where *max-core time* denotes the runtime obtained with the highest number of cores for which the algorithm does not run out of memory. For example, if an algorithm runs out of memory for ≥ 8 cores, max-core time is the runtime obtained for 4 cores.

MC-Join-Clust shows less speedup for higher values of ϵ and BMS-POS (e.g., 8.8 for $\epsilon = 2$ compared to 2.2 for $\epsilon = 5$), CELONIS1 (e.g., 6.2 for $\epsilon = 2$ compared to 4.1 for $\epsilon = 5$), and CELONIS2 (e.g., 9.8 for $\epsilon = 2$ compared to 4.2 for $\epsilon = 5$). For the CELONIS datasets, both MC-Spread and MC-Join-Clust show good runtimes for high values of ϵ and an increasing number of cores with speedups of about 9.3 and 4.3 (CELONIS1, $\epsilon = 3$), and 10.4 and 8.7 (CELONIS2, $\epsilon = 3$), respectively. Despite reaching a plateau, we conclude that MC-Spread shows a higher speedup compared to MC-Join-Clust for all configurations and datasets.

Finally, the clustering thread of MC-Spread incurs some additional overhead due to the density counter updates. However, if we compare Figure 4.1 (Section 4.3.2) and

Figure 4.5, MC-Spread scales considerably better than Simple-MC-Spread for FLICKR and KOSARAK, and retains its good runtime behavior for CELONIS1.

Table 4.1: Speedups (and the corresponding number of cores) for each dataset and $\epsilon \in \{2, 5\}$ (NA ...not available); speedup of algorithm A is $speedup(A) = \frac{\text{1-core time of } A}{\text{max-core time of } A}$.

(a) $\epsilon = 2$ (smallest neighborhoods).

Dataset	MC-Spread		MC-Join-Clust	
	Speedup	**Cores**	**Speedup**	**Cores**
BMS-POS	**10.2**	16	8.8	16
FLICKR	**2.2**	16	NA	NA
KOSARAK	**7.7**	16	3.6	16
LIVEJ	**2.2**	16	1.4	16
ORKUT	**1.6**	16	1.3	16
SPOT	**1.9**	16	1.2	16
CELONIS1	**10.2**	16	6.2	16
CELONIS2	**10.7**	16	9.8	16

(b) $\epsilon = 5$ (largest neighborhoods).

Dataset	MC-Spread		MC-Join-Clust	
	Speedup	**Cores**	**Speedup**	**Cores**
BMS-POS	**5.1**	16	2.2	16
FLICKR	**2.0**	2	NA	NA
KOSARAK	**2.7**	4	NA	NA
LIVEJ	**2.0**	2	NA	NA
ORKUT	**2.0**	2	NA	NA
SPOT	**1.8**	4	NA	NA
CELONIS1	**5.1**	16	4.1	16
CELONIS2	**8.0**	16	4.2	16

MEMORY USAGE Figures 4.8-4.11 show our results for the main memory consumption of MC-Spread and MC-Join-Clust. The following structures are stored on the heap for both algorithms: the collection of sets, the (asymmetric) inverted index, the candidates, and the lookahead neighbors. MC-Join-Clust also stores the materialized ϵ-neighborhoods on the heap. MC-Spread additionally stores the backlinks, the disjoint-set, the lookahead neighborhood array, and the structures used for synchronization (condition variables, atomic counters) on the heap. Interestingly, MC-Join-Clust is more memory efficient than its sequential counterpart and we can compute some data points for which the sequential implementation runs out of memory. The sequential Join-Clust maintains one large array of pairs, *pairs*, whereas MC-Join-Clust maintains $k+1$ individual arrays, $pairs_i$ ($1 \le i \le k+1$) for $k+1$ threads, and $\sum_{i=1}^{k+1} |pairs_i|$ may be smaller than *pairs* (due to different capacities of `std::vector`[24]).

24 https://en.cppreference.com/w/cpp/container/vector

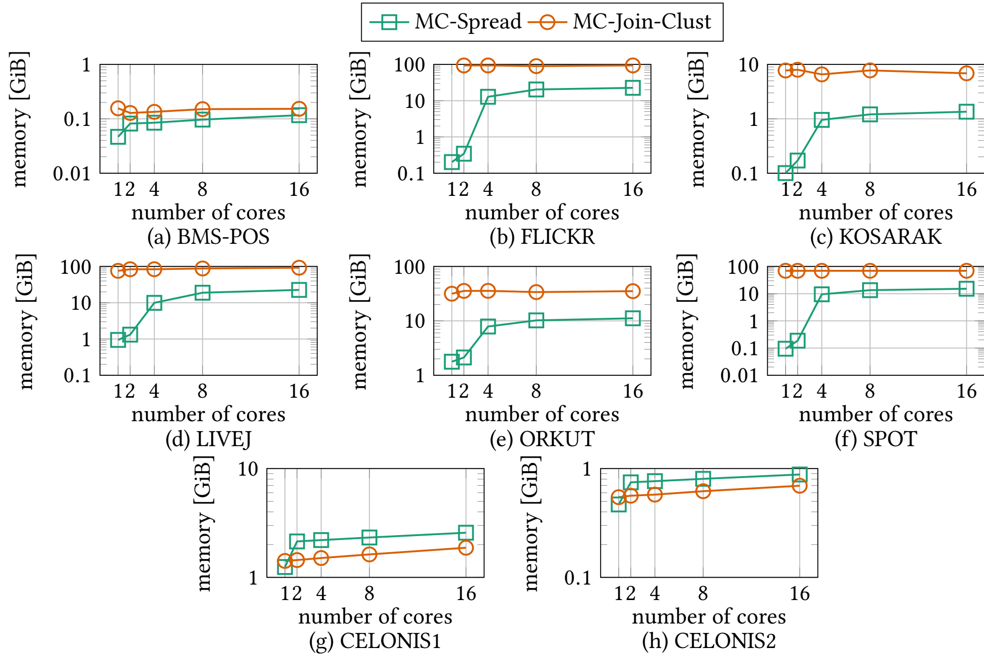


Figure 4.8: Main memory over the number of cores, ϵ = 2, minPts = 16.

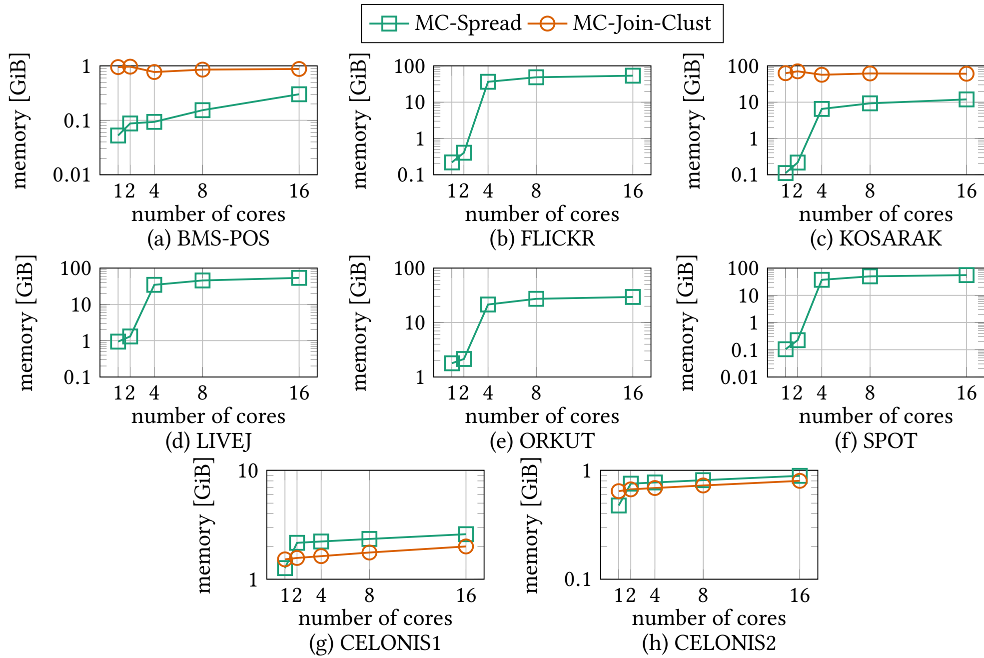


Figure 4.9: Main memory over the number of cores, ϵ = 3, minPts = 16.

Despite small differences, the memory usage of MC-Join-Clust is not sensitive to the number of cores for most datasets. This was to be expected because the materialized neighborhoods dominate the memory consumption of the join-based solution. In contrast, the memory usage of MC-Spread increases with the number of cores, and

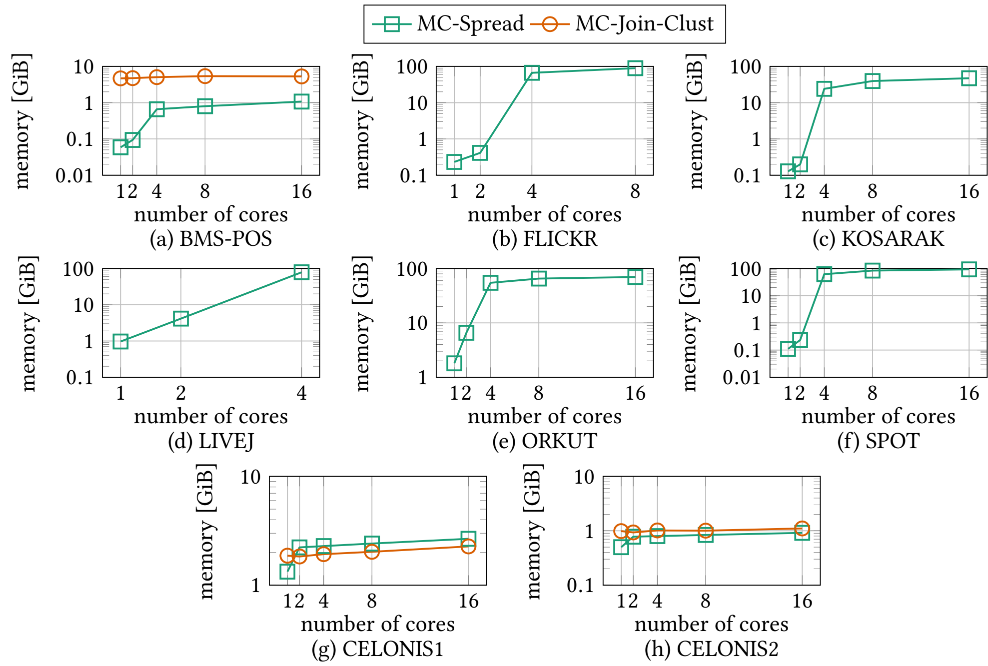


Figure 4.10: Main memory over the number of cores, $\epsilon = 4$, minPts = 16.

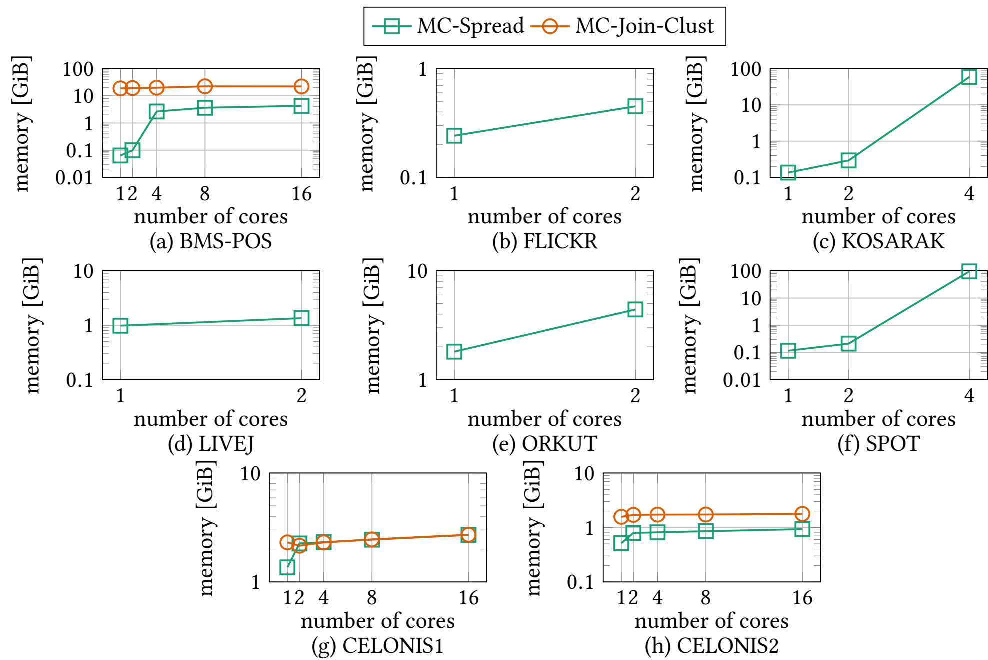


Figure 4.11: Main memory over the number of cores, $\epsilon = 5$, minPts = 16.

MC-Spread runs out of memory for $\epsilon = 5$ on some configurations (FLICKR, LIVEJ, and ORKUT for more than 2 cores; KOSARAK and SPOT for more than 4 cores). A higher number of cores implies more concurrent neighborhood computations and materializations. Consequently, more main memory is consumed since the deallocation

is done by a single clustering thread. One way to mitigate this effect is to limit the number of lookahead neighborhoods that are materialized simultaneously (as described in Section 4.2). In the case of the CELONIS datasets and multiple cores, MC-Spread consumes even more memory than MC-Join-Clust, which is due to the fact that many lookahead neighborhoods are materialized simultaneously. In combination with the other structures (disjoint-set, backlinks, and the synchronization structures), MC-Spread consumes more memory than the (relatively small) neighborhoods of MC-Join-Clust (which maintains no additional structures). For higher values of ϵ this effect is superseded by the neighborhoods that grow larger and dominate the memory.

CACHING We also study the number of CPU cycles and the caching to assess the benefits of our MC-Spread implementation quantitatively (compared to Simple-MC-Spread). For a comparison with Simple-MC-Spread, we refer to Figure 4.2 for the number of CPU cycles and to Figure 4.3 for the cache misses (ϵ = 3, minPts = 16; cf. Section 4.3.2). We compare the CPU cycles and cache misses for the same three datasets (CELONIS1, KOSARAK, and FLICKR) and parameters (ϵ = 3 and minPts = 16). The results are depicted in Figure 4.12 (CPU cycles) and Figure 4.13 (cache misses).

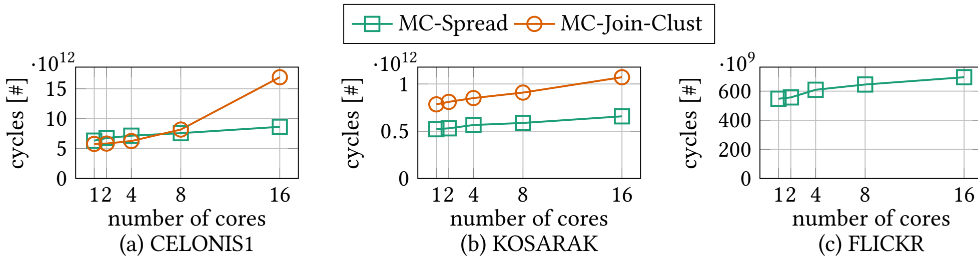


Figure 4.12: CPU cycles for CELONIS1, KOSARAK, and FLICKR, ϵ = 3, minPts = 16.

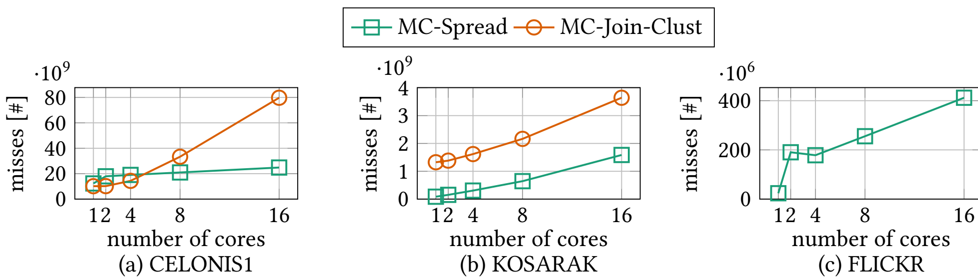


Figure 4.13: Cache misses for CELONIS1, KOSARAK, and FLICKR, ϵ = 3, minPts = 16.

For CELONIS1, the results are similar to the results obtained for Simple-MC-Spread. In the case of KOSARAK and FLICKR, however, we observe that both the number of CPU cycles and the number of cache misses grow at a much lower rate for MC-Spread when we increase the number of cores. For 16 cores and the FLICKR dataset, we measure about an order of magnitude fewer cache misses and about 23x fewer CPU cycles for MC-Spread than for Simple-MC-Spread. This suggests that our adaptations to improve over Simple-MC-Spread work.

CONSTRAINED MEMORY Figures 4.14 (memory consumption) and 4.15 (runtime) show our experimental results obtained for MC-Spread *with* memory constraint (cf. Section 4.3.3). We present our results for $\epsilon = 5$ and the hardest datasets with respect to memory consumption, i.e., FLICKR, KOSARAK, LIVEJ, ORKUT, and SPOT. Recall that the configurations ran out of memory without the memory constraint for more than 2 cores (FLICKR, LIVEJ, and ORKUT) and 4 cores (KOSARAK and SPOT), respectively (cf. Figure 4.11). In our tests, we fix the number of cores to 8, and bound the memory to 8GiB, 16GiB, and 32GiB, respectively.

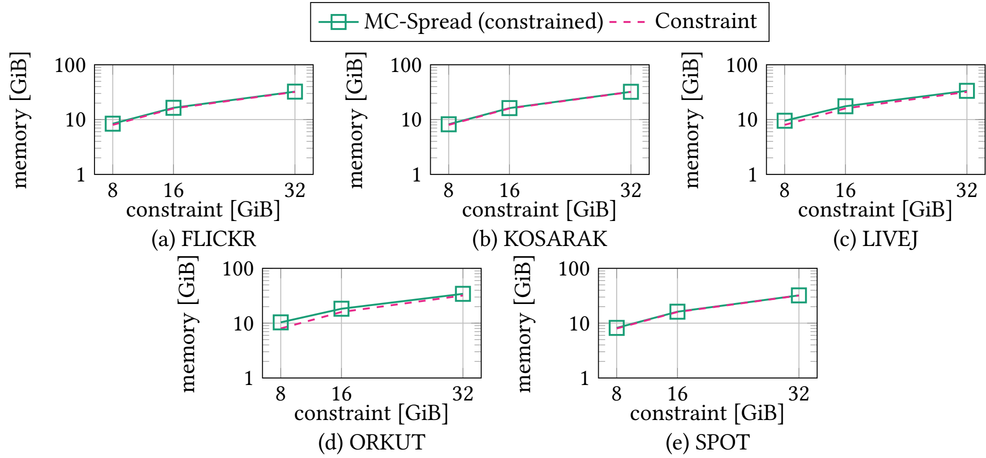


Figure 4.14: Main memory over memory constraint, 8 cores, $\epsilon = 5$, minPts = 16.

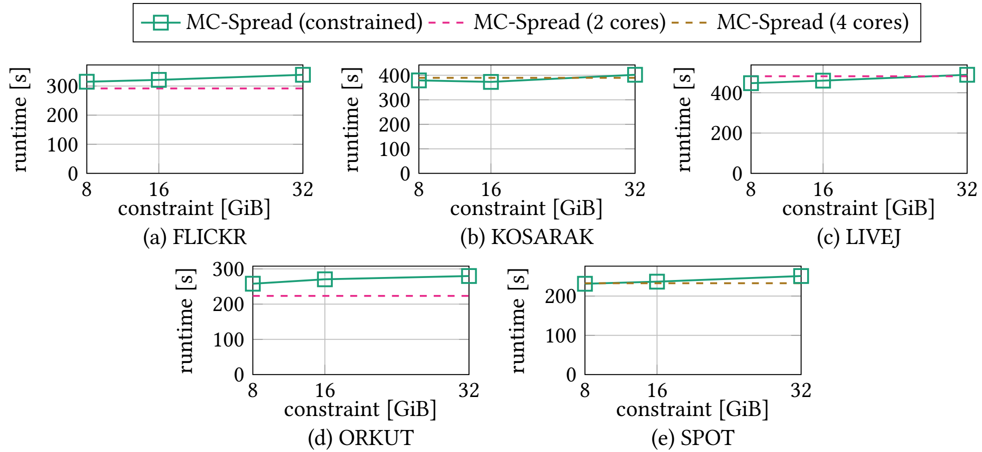


Figure 4.15: Wallclock time over memory constraint, 8 cores, $\epsilon = 5$, minPts = 16.

Our experiments show that bounding the memory allows us to execute all configurations. Figure 4.14 depicts the memory usage over the given constraints. The memory that is allocated by MC-Spread with memory constraint is quite close to the given memory bound (dashed line). This also confirms that the lookahead neighborhoods dominate the memory since our constraint only considers the allocated space of the lookahead neighborhoods. We observe the largest gap (about 2GiB) between consumed

memory and the given bound for a constraint of 8GiB and the ORKUT dataset ($\epsilon = 5$).

Figure 4.15 shows the runtime results for MC-Spread with constrained memory. The dashed lines show the runtimes without memory constraint (cf. Figure 4.7) for the respective configurations using 2 cores (FLICKR, LIVEJ, and ORKUT) and 4 cores (KOSARAK and SPOT), respectively. We observe small differences in runtime. However, bounding the memory does not result in a significant performance degradation for these configurations. Due to the large neighborhoods, the runtime of MC-Spread is dominated by the clustering thread and thus, MC-Spread reaches a plateau for more cores (despite using more memory).

4.7 CONCLUSION & OUTLOOK

In this chapter, we studied the multi-threaded implementation of Spread, MC-Spread. We introduced the Simple-MC-Spread algorithm, which is based on the observation that most of the overall runtime is spent in the neighborhood computation for some configurations. The $k + 1$ threads are split into k threads that compute the lookahead neighborhoods and one thread that clusters the sets into DBSCAN clusters based on the available neighborhoods. The clustering thread also frees the memory of the respective neighborhoods. To prevent the neighborhood threads from filling up the memory (if they are much faster than the clustering thread), we proposed a solution that bounds the memory consumption. We made two observations that limit the scalability of Simple-MC-Spread: (i) Neighborhood threads may be too slow and (ii) the atomic density counters are subject to contention and false sharing. For each observation, we proposed a solution to mitigate the respective limitation. We implemented MC-Spread and compared it against a multi-threaded implementation of Join-Clust, MC-Join-Clust. MC-Join-Clust also parallelizes the neighborhood computation (with $k + 1$ threads) and executes a sequential DBSCAN algorithm based on the materialized neighborhoods. Our experiments revealed that the speedup of both multi-threaded algorithms is limited for some datasets (FLICKR, LIVEJ, ORKUT, and SPOT) due to the large number of (lookahead) neighbors. MC-Join-Clust ran out of memory for many instances of these datasets while MC-Spread was able to compute most of the configurations even without memory constraint (and all configurations with the memory constraint enabled). In terms of runtime, both MC-Join-Clust and MC-Spread reached a plateau for some datasets with large neighborhoods. The number of CPU cycles and cache misses confirmed that MC-Spread benefits from our adaptations with respect to Simple-MC-Spread. Finally, we evaluated the memory-constrained version of MC-Spread and observed that this has only little impact on the runtime in the relevant settings (large neighborhoods) because the clustering thread is the limiting factor.

Outlook

From our experiments, we observe that configurations can be categorized based on the neighborhood computation (cheap or expensive) and the size of the neighborhoods (small or large). (i) Small neighborhoods that are cheap to compute are not problematic.

(ii) If the neighborhoods are small but expensive to compute (e.g., CELONIS1 and CELONIS2), the speed of the neighborhood computations is the limiting factor. Therefore, it is beneficial to have as many threads as possible for the neighborhood computation. (iii) A large neighborhood challenges both MC-Spread and MC-Join-Clust (e.g., for FLICKR, LIVEJ, ORKUT, and SPOT). The join-based solution, MC-Join-Clust, suffers from the expensive materialization and runs out of memory quickly. In MC-Spread, the (single) clustering thread has to do more work than the neighborhood threads (which split the work). As a future research direction, we will aim to move load from the clustering thread to the other threads without worsening the caching of MC-Spread.

Other future work includes (i) pinning threads to sockets and evaluating the memory layout with respect to our architecture (NUMA-awareness) and (ii) to compare against state-of-the-art multi-core DBSCAN algorithms that cluster sets according to our problem definition. Another interesting research path is to solve the set clustering problem in a distributed environment. Remote direct memory access (RDMA) has been successfully used to design scalable distributed index structures and algorithms, including distributed joins [14, 15] and transactions [134], distributed B^+ trees [140] and key-value stores [85], and new schemes for high availability of database systems [135]. Therefore, a distributed algorithm that is designed for fast, RDMA-capable networks could enable scalability beyond the main memory bounds of a single machine.

5 CONCLUSIONS & FUTURE WORK

In this thesis we studied two types of similarity queries: (1) top-k subtree similarity queries for ordered labeled trees and (2) density-based clustering for collections of sets. Instead of equality predicates, a similarity query evaluates similarity predicates, i.e., two data items are compared using a similarity function. We used the similarity function as a black box and focused on the evaluation of the respective query types. We developed specialized index structures and algorithms for the two query types, and empirically evaluated them against state-of-the-art solutions.

The top-k subtree similarity query finds and ranks the k most similar subtrees in a large document tree with respect to a given query tree. We considered ordered labeled trees and used the tree edit distance to assess the similarity between two trees. Previous solutions suffer either from high memory requirements or high runtimes: Index-based solutions build an index to answer a query fast, but the index is quadratic in the input size and does not support updates. In contrast, an index-free solution has a small memory footprint (decoupled from the input size), but is slow because it must scan the entire document tree for each single query. We developed *SlimCone*, an updatable, linear-space index structure that enables us to retrieve promising subtrees first. A subtree is promising if it has many labels with the query tree in common. Our index is based on inverted lists and achieves linear space by avoiding full list materialization. Instead, we build relevant parts of the lists on the fly. Our experiments confirmed the efficiency, the effectiveness, and the memory scalability of our solution. We achieved runtime improvements of up to four orders of magnitude and were able to outperform the index-based state of the art with respect to memory usage, indexing time, and the number of distance computations.

Density-based clustering techniques identify clusters based on the notion of density, i.e., clusters are regions of high density that are separated by regions of lower density. The DBSCAN algorithm is the most popular representative of density-based clustering techniques. It starts with a random data item and recursively expands dense neighborhoods until a neighborhood of low density is encountered. This is done until all data items have been processed. Indexes are used to retrieve the neighbors of a particular data item. We studied density-based clustering in the context of sets under the Hamming distance. The DBSCAN algorithm requires a so-called symmetric index that returns all neighbors of a particular set (independently of the order in which the sets are processed). Unfortunately, the symmetric index is less effective than its asymmetric counterpart that has been developed for set similarity joins. Because the asymmetric index returns only a specific part of the complete neighborhood, the lookahead neighbors, it cannot be readily combined with the DBSCAN algorithm (due to the neighborhood-by-neighborhood processing order). Join-based clustering solutions can use asymmetric indexes but have to materialize all neighborhoods in main memory.

The size of the neighborhoods may be quadratic in the input size, which limits the applicability of this approach. We introduced *Spread*, a DBSCAN-compliant solution for sets, which is able to use asymmetric indexes while only requiring linear space. To this end, we impose a processing order and only materialize a single lookahead neighborhood at a time. So-called backlinks store sufficient information to derive a correct clustering despite the usage of asymmetric indexes. Our experiments suggest that Spread is competitive with the join-based solution in terms of runtime while retaining the memory efficiency of the DBSCAN algorithm.

Finally, we studied *MC-Spread*, a multi-core extension of our single-core solution for the density-based clustering of sets. We presented an approach to parallelize Spread by interleaving neighborhood computation and clustering: All but one thread are used to compute (and materialize) lookahead neighborhoods, and a single thread is used to build the clusters. We proposed solutions for cache locality and load balancing issues that arise in the multi-core implementation of Spread. We implemented MC-Spread as well as a multi-threaded version of the join-based solution, MC-Join-Clust, and evaluated them experimentally for a varying number of cores. Our experiments suggest that both algorithms scale well with the number of cores for some datasets (with small neighborhoods that are expensive to compute). For datasets with very large neighborhoods, the sequential clustering thread dominates the runtime and limits scalability.

Future Work

Our solution to answer top-k subtree similarity queries, SlimCone, uses lower bounds on the size and the labels of the trees. It would be interesting to include a lower bound that considers the structure of the trees as well. If integrated carefully into our index structure, this could further improve the performance due to the additional pruning power. Furthermore, SlimCone is designed as a single-core algorithm. Extending SlimCone to multi-core and/or distributed environments would be another interesting research direction. A starting point are the partitions obtained from the size lower bound that promise to allow parallelization with little synchronization. Finally, adapting SlimCone to answer top-k subtree similarity queries for unordered trees (for which computing the tree edit distance has been shown to be NP-complete) could be another future research direction.

We designed our solution for the density-based set clustering problem, Spread, as a single-core algorithm. In practice, an extension to multi-core and/or distributed environments is desirable. Chapter 4 presented a multi-core extension for density-based clustering of sets, MC-Spread, that is compared against a multi-core version of the join-based approach, MC-Join-Clust. It would be interesting to empirically compare MC-Spread also to other parallel DBSCAN solutions. Another interesting research direction is to control the memory consumption of MC-Spread. Finally, it would be interesting to evaluate Spread and MC-Spread on additional similarity functions and asymmetric indexes.

For both types of similarity queries, top-k subtree similarity queries and density-

based clustering of sets, it would be interesting to design and evaluate solutions for distributed environments with modern networks. Apart from providing high throughput and low latency, modern networking hardware also supports remote direct memory access (RDMA). In a distributed environment, RDMA allows a machine to directly access the main memory of another machine (bypassing the operating system kernel and the CPU). We see an opportunity that carefully designed algorithms for RDMA-capable networks could enable solutions that scale beyond the main memory bounds of a single machine.

REPRODUCIBILITY PACKAGE

A.1 HARDWARE, OPERATING SYSTEM, AND SOFTWARE

All experiments were tested on a 64-bit machine with

- 2 physical processors, Intel(R) Xeon(R) CPUs E5-2630 v3 2.40 GHz,
- 8 cores per physical processor (⇒ 16 logical processors),
- 3 cache levels with sizes 32 KiB (L1d), 32 KiB (L1i), 256 KiB (L2), and 20.480 KiB (L3),
- 96 GiB of main memory @ 2.133 MHz (1.866 MHz configured clock speed),
- 2x 1.8 TiB HDDs as secondary storage with a theoretical performance of (1) *read (cache miss/hit)*: 0,5/0,1ms, (2) *write*: 0,015ms, (3) *seek*: 0,5ms,
- Debian 9 Stretch (`Linux 4.9.0-8-amd64 #1 SMP Debian 4.9.144-3 (2019-02-02) x86_64`) as OS, and
- the following software packages installed:
  - `ansible` [1] (version ≥ 2.2.1.0)
  - `wget` (version ≥ 1.18) and `tar` (version ≥ 1.29)

We expect the experiments to run on any machine with modern hardware and the abovementioned versions of Debian Linux and Ansible installed. Ansible will install all additional software packages (using `apt`).

A.2 QUICK START

Open a terminal and follow three steps:

1. Install Ansible, wget, and tar

```
sudo apt-get install ansible wget tar # requires sudo/root permissions
```

2. Download and extract reproducibility package [2]

```
wget https://kitten.cosy.sbg.ac.at/index.php/s/fjT3eQ76JekgAK3/download \
  -O sigmod2019-reproducibility.tar.gz
tar xzvf sigmod2019-reproducibility.tar.gz
```

3. Run the main Ansible playbook

```
cd sigmod2019-reproducibility
# asks for sudo password to install packages (hit "Enter" if you are root)
ansible-playbook -K run_all.yaml
```

1 https://www.ansible.com/
2 https://kitten.cosy.sbg.ac.at/index.php/s/fjT3eQ76JekgAK3

A.3 REPRODUCIBILITY PACKAGE

The main Ansible playbook `run_all.yaml` will automatically (a) install all required software packages, (b) download and extract datasets [3] (2.2 GiB) and queries [4] (76 KiB), (c) compile the C++ source code, (d) set up and run all experiments, (e) extract the raw experimental results, and (f) compile the paper with the obtained results.

A.3.1 *Datasets, Queries, and Results*

Both datasets and queries contain two directories `xmark/` and `realworld/`. The `xmark/` directory contains five synthetic datasets (generated using the XMark benchmark) and for each XMark dataset we extracted four queries with 4, 8, 16, 32, and 64 nodes, respectively (i.e., 100 queries in total). Similarly, the `realworld/` directory contains three real-world datasets (TreeBank, DBLP, and SwissProt) and for each real-world dataset we extracted four queries with 4, 8, 16, 32, and 64 nodes, respectively (i.e., 60 queries in total). Queries were extracted from the corresponding datasets. Naming example: `xmark4_query_16_2.xml` is the second query with 16 nodes that was extracted from the XMark4 dataset and will be used for this dataset in our experiments.

By default, all experimental results are written into a directory `results/`, which is created automatically. For each of the following plots a dedicated subdirectory is created in `results/`:

- Figure 12: `fig12_ab/` `fig12_cd/` `fig12_e/` `fig12_f/`
- Figure 13: `fig13_ab/` `fig13_cd/`
- Figure 14: `fig14_ab/` `fig14_cd/`
- Figure 15: `fig15_ab/` `fig15_cd/`
- Figure 16: `fig16_ab/` `fig16_cd/`

This naming convention for subdirectories is also used in the source code directory of our paper: The directory `paper/figs/experiments/` contains the `pgfplots` files, and the corresponding result files can be found in `paper/csv/` (copied from `results/`).

A.3.2 *Package Details*

The package contains several Python (`py`) and Ansible (`yaml`) files. Further, the C++ source code for all algorithms (directories `tasm-struct/` and `slim/`) as well as the paper's source code (directory `paper/`) are provided. Finally, some directories are generated during execution. This section summarizes the most important parts and behaviors.

ANSIBLE We use Ansible to automate our experiments, i.e., the main pipeline to produce the results of our paper is executed using Ansible. Ansible uses `yaml` files and the purpose of the respective `yaml` files can be summarized as follows. `run_all.yaml`

3 https://kitten.cosy.sbg.ac.at/index.php/s/jYJC2xzPCNnjJZD
4 https://kitten.cosy.sbg.ac.at/index.php/s/m4JixE8xXKG7xkM

installs all required software packages and executes all other Ansible (`yaml`) files (in this order):

`get_data.yaml` Fetches the datasets and queries used for our experiments.

`build_code.yaml` Automatically builds the C++ source code for the algorithms TASM (TASM-Postorder [8]), STRUCT (StructureSearch [31]), and our algorithms MERGE, CONE, SLIM, and SLIM-DYN [69].

`build_symlinks.yaml` Creates the subdirectories for all figures and the symbolic links to the datasets and queries of the respective figures. For example, `datasets/fig13_ab/` contains symbolic links to all XMark datasets and `queries/fig13_ab/` contains symbolic links to all XMark queries with $|Q| = 16$ nodes.

`run_experiments.yaml` Executes experiments for a specific configuration of figure number, number of runs and simultaneously executed processes, a list of k values, a default k value (for experiments with fixed k), and a list of update counts (repeated calls of `profile-all.py`, see below).

`copy_result_files.yaml` Copies result files from `results/` to `paper/csv/`.

Afterwards, the paper is compiled (`Makefile`) and the resulting PDF file can be found in `paper/paper.pdf`.

The following parameters can be passed to `run_all.yaml` using JSON syntax:

`"vary_k"` Space-separated string of integers for varying-k experiments (default: “`1 10 100`”).

`"default_k"` Single integer that is used for experiments with a fixed k (default: `10`).

`"runs"` Number of runs per experiments (robustness parameter; default: `1`).

`"processes"` Number of simultaneously executed processes (robustness parameter; default: `1`).

`"updates"` Space-separated string of integers for update experiments (default: “`1 10 100 ... 1000000`”).

Example: $k \in \{2, 4, 6\}$, fixed $k = 5$, and 7 simultaneously executed processes

```
1 ansible-playbook -K run_all.yaml --extra-vars= \
2   '{ "vary_k": "2 4 6", "default_k": 5, "processes": 7 }'
```

PYTHON We use Python (`python3`) to (1) run the experiments, (2) extract the required statistics from the experimental results, and (3) create the result files s.t. they can be used in our paper. Run scripts with “`--help`” to show all options.

`profile-all.py` Calls `profile.py` repeatedly to run all experiments for figures 12–16.

`profile.py` Executes experiments for a given list of k values, a directory of XML datasets, a directory of XML queries using a given algorithm (C++ binary). The results are stored in a Python dictionary that is serialized to disk. To speed up the experiments, the number of simultaneously executed processes can be specified using “`--maxprocesses`” (default: 1; max. robustness).

`mystatistics.py` Extracts *all* statistics from a given serialized dictionary to seperate `stat` files (in CSV format), one for each statistics entry found in the dictionary. Typically, this script is executed for each single serialized dictionary produced by `profile.py`. Adaptations may be necessary if the experiments are changed.

`merge.py/replaceheader.py/suffixcolumnorder.py` Helper scripts that merge multiple `stat` files and postprocess them s.t. they can be used in our paper plots (which use `pgfplots`). Adaptations may be necessary if the experiments are changed.

DIRECTORY STRUCTURE OF THE PACKAGE

`common/` Helper files that are used by `profile.py`.

`paper/` LaTeX source code of our paper. Compile with `make`.

`slim/` Source code of our algorithms MERGE, CONE, SLIM, and SLIM-DYN [69]. Build with `-DNO_LOG` and `-DNO_INALGO_TIMINGS`, i.e., using `cmake -D CMAKE_CXX_FLAGS="-DNO_LOG -DNO_INALGO_TIMINGS"`.

`tasm-struct/` Source code of the competitor algorithms TASM (TASM-Postorder [8]) and STRUCT (StructureSearch [31]). Build with `-DNO_INALGO_TIMINGS`, i.e., using `cmake -D CMAKE_CXX_FLAGS="-DNO_INALGO_TIMINGS"`.

DIRECTORIES CREATED DURING EXECUTION

`datasets/` All datasets, fetched automatically by the Ansible script `run_all.yaml`.

`queries/` All queries, fetched automatically by the Ansible script `run_all.yaml`.

`tmp/` The C++ binary to be executed is copied into this directory for the experiments.

`results/` All raw exp. results and result files for our paper plots. Subdirectory naming is discussed in Section A.3.1.

A.4 FLEXIBILITY

A.4.1 *Parameters*

Relevant parameters for our research problem are the dataset (document), the query, and the result size k. We consider a broad range for each single parameter in our paper, i.e., we varied dataset sizes (synthetic and real-world, ranging from 83 MiB to 6.1 GiB), query sizes (five query sizes for each dataset, 4 – 64 nodes), and result size k (range: 1, 10, $\ldots 10^4$). We track the build time and the size of the index, the query time, and the number of verified subtrees.

Additional datasets and queries can be included into the `datasets/` and `queries/` directories, respectively. Note that both need to be XML files and that the filename of the query must have the dataset name as a prefix. Only if this is the case, the script `profile.py` will use the query as input for the respective dataset. Finally, the symbolic link to the dataset resp. query in the figure subdirectory of the `dataset/` resp. `queries/` directory must be created. For example, to include a new dataset `abc.xml`

and a new query abc_query.xml (note the prefix of the query filename) into Figure 12a, two symbolic links are required:

- dataset/fig12_ab/abc.xml → abc.xml
- queries/fig12_ab/abc_query.xml → abc_query.xml

To test other values of k, the default setup is changed as in the following example, where $k \in \{2, 4, 8, 16, 32\}$:

```
1 ansible-playbook -K run_all.yaml --extra-vars= '{ "vary_k": "2 4 8 16 32" }'
```

A.4.2 *Plots*

We use pgfplots to generate our plots. We tailored the plot configurations towards the specific results presented in our paper. To include additional columns or change the x-/y-range, the pgfplots files need to be changed accordingly. These files are stored in paper/figs/experiments/. However, it should still be possible to extract the raw experimental results (using profile-all.py) for new datasets, queries, and k values.

A.5 TIME ESTIMATES

The reproducibility pipeline is not optimized towards minimizing the number of executions, i.e., some parameter configurations may be executed multiple times. This is to simplify the structure of profile-all.py, which is organized in a figure-based manner. Based on our experiences, running all experiments with 2 simultaneously executed processes and 1 run will take about 40 hours.

Our estimations assume that a single run is executed per experiment. For the experiments in our paper, we use 5 runs per experiment and compute the average over all runs. The runtimes can be reduced by executing processes in parallel. However, increasing the number of parallel processes may affect the robustness of the results. For example, to use 6 parallel processes, call the main Ansible file as follows:

```
1 ansible-playbook -K run_all.yaml --extra-vars='{ "processes": 6 }'
```

[1] Wil van der Aalst. *Process Mining: Data Science in Action.* 2nd. Springer Publishing Company, Incorporated, 2016. ISBN: 3662498502.

[2] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. "Best Position Algorithms for Top-k Queries". In: *Proceedings of the 33rd International Conference on Very Large Data Bases.* VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 495–506. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325909.

[3] Tatsuya Akutsu. "Tree Edit Distance Problems: Algorithms and Applications to Bioinformatics". In: *IEICE Transactions on Information and Systems* E93.D.2 (2010), pp. 208–218.

[4] Sattam Alsubaiee et al. "AsterixDB: A Scalable, Open Source BDMS". In: *Proc. VLDB Endow.* 7.14 (Oct. 2014), 1905–1916. ISSN: 2150-8097. DOI: 10.14778/2733085.2733096. URL: https://doi.org/10.14778/2733085.2733096.

[5] Kiyoko F Aoki, Atsuko Yamaguchi, Yasushi Okuno, Tatsuya Akutsu, Nobuhisa Ueda, Minoru Kanehisa, and Hiroshi Mamitsuka. "Efficient tree-matching methods for accurate carbohydrate database queries". In: *Genome Informatics* 14 (2003), pp. 134–143.

[6] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. "Efficient Exact Set-Similarity Joins". In: *Proceedings of the 32nd International Conference on Very Large Data Bases.* VLDB '06. Seoul, Korea: VLDB Endowment, 2006, 918–929.

[7] James Archibald and Jean-Loup Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". In: *ACM Trans. Comput. Syst.* 4.4 (Sept. 1986), 273–298. ISSN: 0734-2071. DOI: 10.1145/6513.6514. URL: https://doi.org/10.1145/6513.6514.

[8] N. Augsten, D. Barbosa, M. Böhlen, and T. Palpanas. "TASM: Top-k Approximate Subtree Matching". In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010).* 2010, pp. 353–364. DOI: 10.1109/ICDE.2010.5447905.

[9] N. Augsten, D. Barbosa, M. Bohlen, and T. Palpanas. "Efficient Top-k Approximate Subtree Matching in Small Memory". In: *IEEE Transactions on Knowledge and Data Engineering* 23.8 (2011), pp. 1123–1137. ISSN: 1041-4347. DOI: 10.1109/TKDE.2010.245.

[10] Nikolaus Augsten. "A Roadmap towards Declarative Similarity Queries". In: *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.* Ed. by Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose. OpenProceedings.org, 2018, pp. 509–512. DOI: 10.5441/002/edbt.2018.59. URL: https://doi.org/10.5441/002/edbt.2018.59.

[11] Nikolaus Augsten and Michael H. Böhlen. *Similarity Joins in Relational Database Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013. ISBN: 9781627050289. DOI: 10.2200/S00544ED1V01Y201310DTM038. URL: http://dx.doi.org/10.2200/S00544ED1V01Y201310DTM038.

[12] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. "Approximate Matching of Hierarchical Data Using pq-grams". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 301–312. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083630.

[13] Nikolaus Augsten, Michael Böhlen, Curtis Dyreson, and Johann Gamper. "Windowed pq-grams for approximate joins of data-centric XML". In: *The VLDB Journal* 21.4 (2012), pp. 463–488. ISSN: 0949-877X. DOI: 10.1007/s00778-011-0254-6. URL: https://doi.org/10.1007/s00778-011-0254-6.

[14] Claude Barthels. "Scalable Query and Transaction Processing over High-Performance Networks". In: 2019.

[15] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. "Rack-Scale In-Memory Join Processing Using RDMA". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1463–1475. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2750547. URL: http://doi.acm.org/10.1145/2723372.2750547.

[16] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. "Scaling Up All Pairs Similarity Search". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 131–140. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242591. URL: http://doi.acm.org/10.1145/1242572.1242591.

[17] J. Bellando and R. Kothari. "Region-based modeling and tree edit distance as a basis for gesture recognition". In: *Proceedings 10th International Conference on Image Analysis and Processing*. 1999, pp. 698–703. DOI: 10.1109/ICIAP.1999.797676.

[18] Philip Bille. "A survey on tree edit distance and related problems". In: *Theoretical Computer Science* 337.1 (2005), pp. 217 –239. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2004.12.030. URL: http://www.sciencedirect.com/science/article/pii/S0304397505000174.

[19] G. Blin, A. Denise, S. Dulucq, C. Herrbach, and H. Touzet. "Alignments of RNA Structures". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 7.2 (2010), pp. 309–322. DOI: 10.1109/TCBB.2008.28.

[20] Christian Böhm, Bernhard Braunmüller, Markus Breunig, and Hans-Peter Kriegel. "High Performance Clustering Based on the Similarity Join". In: *Proceedings of the Ninth International Conference on Information and Knowledge Management*. CIKM '00. McLean, Virginia, USA: Association for Computing

Machinery, 2000, 298–305. ISBN: 1581133200. DOI: 10.1145/354756.354832. URL: https://doi.org/10.1145/354756.354832.

[21] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. "Density-Based Clustering Using Graphics Processors". In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management.* CIKM '09. Hong Kong, China: Association for Computing Machinery, 2009, 661–670. ISBN: 9781605585123. DOI: 10.1145/1645953.1646038. URL: https://doi.org/10.1145/1645953.1646038.

[22] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. "Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models". In: *Business Process Management Workshops.* Ed. by Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 170–181. ISBN: 978-3-642-12186-9.

[23] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. "Spatio-Textual Similarity Joins". In: *Proc. of the VLDB Endowment* 6.1 (Nov. 2012), 1–12.

[24] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://rfc-editor.org/rfc/rfc8259.txt.

[25] S. Brecheisen, H. . Kriegel, and M. Pfeifle. "Efficient Density-Based Clustering of Complex Objects". In: *Fourth IEEE International Conference on Data Mining (ICDM'04).* 2004, pp. 43–50.

[26] Pável Calado, Melanie Herschel, and Luís Leitão. "An Overview of XML Duplicate Detection Algorithms". In: *Soft Computing in XML Data Management - Intelligent Systems from Decision Making to Data Mining, Web Intelligence and Computer Vision.* Vol. 255. Studies in Fuzziness and Soft Computing. Springer, 2010, pp. 193–224. ISBN: 978-3-642-14009-9. DOI: 10.1007/978-3-642-14010-5_8.

[27] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. "Better bitmap performance with Roaring bitmaps". In: *Software: Practice and Experience* 46.5 (2015), 709–719. ISSN: 0038-0644. DOI: 10.1002/spe.2325. URL: http://dx.doi.org/10.1002/spe.2325.

[28] Samy Chambi, Daniel Lemire, Robert Godin, Kamel Boukhalfa, Charles R. Allen, and Fangjin Yang. "Optimizing Druid with Roaring Bitmaps". In: *Proceedings of the 20th International Database Engineering & Applications Symposium.* IDEAS '16. Montreal, QC, Canada: Association for Computing Machinery, 2016, 77–86. ISBN: 9781450341189. DOI: 10.1145/2938503.2938515. URL: https://doi.org/10.1145/2938503.2938515.

[29] S. Chaudhuri, V. Ganti, and R. Kaushik. "A Primitive Operator for Similarity Joins in Data Cleaning". In: *22nd International Conference on Data Engineering (ICDE'06).* 2006, pp. 5–5.

[30] S. Cohen and N. Or. "A General Algorithm for Subtree Similarity-Search". In: *2014 IEEE 30th International Conference on Data Engineering*. 2014, pp. 928–939. DOI: 10.1109/ICDE.2014.6816712.

[31] Sara Cohen. "Indexing for Subtree Similarity-Search Using Edit Distance". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 49–60. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2463716. URL: http://doi.acm.org/10.1145/2463676.2463716.

[32] I. Cordova and T. Moh. "DBSCAN on Resilient Distributed Datasets". In: *2015 International Conference on High Performance Computing Simulation (HPCS)*. 2015, pp. 531–540.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[34] B. Dai and I. Lin. "Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition". In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 59–66.

[35] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. "An optimal decomposition algorithm for tree edit distance". In: *ACM Transactions on Algorithms* 6.1 (2009).

[36] Dong Deng, Yufei Tao, and Guoliang Li. "Overlap Set Similarity Joins with Theoretical Guarantees". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, 905–920. ISBN: 9781450347037. DOI: 10.1145/3183713.3183748. URL: https://doi.org/10.1145/3183713.3183748.

[37] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. "An Efficient Partition Based Method for Exact Set Similarity Joins". In: *Proc. VLDB Endow.* 9.4 (Dec. 2015), pp. 360–371. ISSN: 2150-8097. DOI: 10.14778/2856318.2856330. URL: http://dx.doi.org/10.14778/2856318.2856330.

[38] Lee R. Dice. "Measures of the Amount of Ecologic Association Between Species". In: *Ecology* 26.3 (1945), pp. 297–302. DOI: https://doi.org/10.2307/1932409. eprint: https://esajournals.onlinelibrary.wiley.com/doi/pdf/10.2307/1932409. URL: https://esajournals.onlinelibrary.wiley.com/doi/abs/10.2307/1932409.

[39] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. 7th. Pearson, 2015. ISBN: 0133970779.

[40] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231. URL: http://dl.acm.org/citation.cfm?id=3001460.3001507.

[41] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. “Incremental Clustering for Mining in a Data Warehousing Environment”. In: *Proceedings of the 24rd International Conference on Very Large Data Bases*. VLDB ’98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, 323–333. ISBN: 1558605665.

[42] Ronald Fagin, Amnon Lotem, and Moni Naor. “Optimal Aggregation Algorithms for Middleware”. In: *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’01. Santa Barbara, California, USA: ACM, 2001, pp. 102–113. ISBN: 1-58113-361-8. DOI: 10.1145/375551.375567. URL: http://doi.acm.org/10.1145/375551.375567.

[43] Ronald Fagin, Amnon Lotem, and Moni Naor. “Optimal Aggregation Algorithms for Middleware”. In: *J. Comput. Syst. Sci.* 66.4 (June 2003), pp. 614–656. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(03)00026-6. URL: http://dx.doi.org/10.1016/S0022-0000(03)00026-6.

[44] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. “Fine-grained and Accurate Source Code Differencing”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 313–324. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642982. URL: http://doi.acm.org/10.1145/2642937.2642982.

[45] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. “Set Similarity Joins on Mapreduce: An Experimental Survey”. In: *Proc. VLDB Endow.* 11.10 (June 2018), pp. 1110–1122. ISSN: 2150-8097. DOI: 10.14778/3231751.3231760. URL: https://doi.org/10.14778/3231751.3231760.

[46] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Franz Färber, and Norman May. “DeltaNI: An Efficient Labeling Scheme for Versioned Hierarchical Data”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: ACM, 2013, pp. 905–916. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465329. URL: http://doi.acm.org/10.1145/2463676.2465329.

[47] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Faerber. “Indexing Highly Dynamic Hierarchical Data”. In: *Proc. VLDB Endow.* 8.10 (June 2015), 986–997. ISSN: 2150-8097. DOI: 10.14778/2794367.2794369. URL: https://doi.org/10.14778/2794367.2794369.

[48] Junhao Gan and Yufei Tao. “DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 519–530. ISBN: 9781450327589. DOI: 10.1145/2723372.2737792. URL: https://doi.org/10.1145/2723372.2737792.

[49] Junhao Gan and Yufei Tao. "Dynamic Density Based Clustering". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 1493–1507. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064050. URL: http://doi.acm.org/10.1145/3035918.3064050.

[50] Markus Götz, Christian Bodenstein, and Morris Riedel. "HPDBSCAN: Highly Parallel DBSCAN". In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. MLHPC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340069. DOI: 10.1145/2834892.2834894. URL: https://doi.org/10.1145/2834892.2834894.

[51] Torsten Grust. "Accelerating XPath Location Steps". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD '02. Madison, Wisconsin: ACM, 2002, pp. 109–120. ISBN: 1-58113-497-5. DOI: 10.1145/564691.564705. URL: http://doi.acm.org/10.1145/564691.564705.

[52] Richard W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[53] D. Han, A. Agrawal, W. Liao, and A. Choudhary. "A Novel Scalable DBSCAN Algorithm with Spark". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 1393–1402.

[54] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. "MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 2011, pp. 473–480.

[55] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. "MR-DBSCAN: A Scalable MapReduce-Based DBSCAN Algorithm for Heavily Skewed Data". In: *Front. Comput. Sci.* 8.1 (Feb. 2014), 83–99. ISSN: 2095-2228. DOI: 10.1007/s11704-013-3158-3. URL: https://doi.org/10.1007/s11704-013-3158-3.

[56] Claire Herrbach, Alain Denise, and Serge Dulucq. "Average complexity of the Jiang-Wang-Zhang pairwise tree alignment algorithm and of a RNA secondary structure alignment algorithm". In: *Theoretical Computer Science* 411.26 (2010), pp. 2423 –2432. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2010.01.014. URL: http://www.sciencedirect.com/science/article/pii/S0304397510000393.

[57] Holger Heumann and Gabriel Wittum. "The tree-edit-distance, a measure for quantifying neuronal morphology". In: *Neuroinformatics* 7.3 (2009), pp. 179–190.

[58] B.F.A. Hompes, J.C.A.M. Buijs, W.M.P. van der Aalst, P.M. Dixit, and J. Buurman. "Discovering Deviating Cases and Process Variants Using Trace Clustering". English. In: *27th Benelux Conference on Artificial Intelligence, 5-6 November 2015, Hasselt, Belgium*. 27th Benelux Conference on Artificial Intelligence (BNAIC

2015), BNAIC 2015 ; Conference date: 05-11-2015 Through 06-11-2015. 2015. URL: http://bnaic2015.org/.

[59] T. Hütter, M. Pawlik, R. Löschinger, and N. Augsten. "Effective Filters and Linear Time Verification for Tree Similarity Joins". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE).* 2019, pp. 854–865. DOI: 10.1109/ICDE.2019.00081.

[60] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. "A Survey of Top-*k* Query Processing Techniques in Relational Database Systems". In: *ACM Comput. Surv.* 40.4 (2008), 11:1–11:58. DOI: 10.1145/1391729.1391730. URL: http://doi.acm.org/10.1145/1391729.1391730.

[61] Paul Jaccard. *Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines.* Bulletin de la Société Vaudoise des Sciences Naturelles: Société Vaudoise des Sciences Naturelles. Rouge, 1901.

[62] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123797519.

[63] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. "Scalable Density-Based Distributed Clustering". In: *Knowledge Discovery in Databases: PKDD 2004.* Ed. by Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 231–244. ISBN: 978-3-540-30116-5.

[64] Heinrich Jiang, Jennifer Jang, and Jakub Lacki. *Faster DBSCAN via subsampled similarity queries.* 2020. arXiv: 2006.06743 [cs.LG].

[65] Mohamed Kashkoush and Hoda ElMaraghy. "Matching Bills of Materials Using Tree Reconciliation". In: *Procedia CIRP* 7 (2013). Forty Sixth CIRP Conference on Manufacturing Systems 2013, pp. 169 –174. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2013.05.029. URL: http://www.sciencedirect.com/science/article/pii/S2212827113002369.

[66] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. "On the Integration of Structure Indexes and Inverted Lists". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data.* SIGMOD '04. Paris, France: ACM, 2004, pp. 779–790. ISBN: 1-58113-859-8. DOI: 10.1145/1007568.1007656. URL: http://doi.acm.org/10.1145/1007568.1007656.

[67] Yeonjung Kim, Jeahyun Park, Taehwan Kim, and Joongmin Choi. "Web information extraction by HTML tree edit distance matching". In: *2007 International Conference on Convergence Information Technology (ICCIT 2007).* IEEE. 2007, pp. 2455–2460.

[68] Philip Klein, Srikanta Tirthapura, Daniel Sharvit, and Ben Kimia. "A tree-edit-distance algorithm for comparing simple, closed shapes". In: *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms.* 2000, pp. 696–704.

[69] Daniel Kocher and Nikolaus Augsten. "A Scalable Index for Top-k Subtree Similarity Queries". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, 1624–1641. ISBN: 9781450356435. DOI: 10.1145/3299869.3319892. URL: https://doi.org/10.1145/3299869.3319892.

[70] Daniel Kocher, Nikolaus Augsten, and Willi Mann. "Scaling Density-Based Clustering to Large Collections of Sets". In: *Proceedings of the 24rd International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*. OpenProceedings.org, 2021. ISBN: 978-3-89318-084-4.

[71] Marzena Kryszkiewicz and Piotr Lasek. "TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality". In: *Rough Sets and Current Trends in Computing*. Ed. by Marcin Szczuka, Marzena Kryszkiewicz, Sheela Ramanna, Richard Jensen, and Qinghua Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 60–69. ISBN: 978-3-642-13529-3.

[72] K. Mahesh Kumar and A. Rama Mohan Reddy. "A fast DBSCAN clustering algorithm by accelerating neighbor searching using Groups method". In: *Pattern Recognition* 58 (2016), pp. 39 –48. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2016.03.008. URL: http://www.sciencedirect.com/science/article/pii/S0031320316001035.

[73] Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Tree-Encoded Bitmaps". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 937–967. ISBN: 9781450367356. DOI: 10.1145/3318464.3380588. URL: https://doi.org/10.1145/3318464.3380588.

[74] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. "Consistently faster and smaller compressed bitmaps with Roaring". In: *Software: Practice and Experience* 46.11 (2016), pp. 1547–1569. DOI: https://doi.org/10.1002/spe.2402. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2402. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2402.

[75] I. Vladimir Levenshtein. "Binary codes capable of correcting spurious insertions and deletions of ones". In: *Problems of Information Transmission* 1 (1965), pp. 8–17.

[76] Fei Li, Hongzhi Wang, Jianzhong Li, and Hong Gao. "A Survey on Tree Edit Distance Lower Bound Estimation Techniques for Similarity Join on XML Data". In: *SIGMOD Rec.* 42.4 (Feb. 2014), pp. 29–39. ISSN: 0163-5808. DOI: 10.1145/2590989.2590994. URL: http://doi.acm.org/10.1145/2590989.2590994.

[77] Zhiwei Lin, Hui Wang, and Sally McClean. "Measuring Tree Similarity for Natural Language Processing Based Information Retrieval". In: *Natural Language Processing and Information Systems*. Ed. by Christina J. Hopfe, Yacine Rezgui, Elisabeth Métais, Alun Preece, and Haijiang Li. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 13–23. ISBN: 978-3-642-13881-2.

[78] A. X. Liu, K. Shen, and E. Torng. "Large scale Hamming distance query processing". In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 553–564. DOI: 10.1109/ICDE.2011.5767831.

[79] Yinghua Lv, Tinghuai Ma, Meili Tang, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. "An efficient and scalable density-based clustering algorithm for datasets with complex structures". In: *Neurocomputing* 171 (2016), pp. 9 –22. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2015.05.109. URL: http://www.sciencedirect.com/science/article/pii/S0925231215008073.

[80] Willi Mann and Nikolaus Augsten. "PEL: Position-Enhanced Length Filter for Set Similarity Joins". In: *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken, Bozen-Bolzano, Italy, October 21st to 24th, 2014*. Ed. by Friederike Klan, Günther Specht, and Hans Gamper. Vol. 1313. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 89–94. URL: http://ceur-ws.org/Vol-1313/paper_16.pdf.

[81] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. "An Empirical Evaluation of Set Similarity Join Techniques". In: *Proc. VLDB Endow.* 9.9 (May 2016), pp. 636–647. ISSN: 2150-8097. DOI: 10.14778/2947618.2947620. URL: http://dx.doi.org/10.14778/2947618.2947620.

[82] Willi Mann, Nikolaus Augsten, and Christian S. Jensen. "SWOOP: Top-k Similarity Joins over Set Streams". In: *CoRR* abs/1711.02476 (2017). arXiv: 1711.02476. URL: http://arxiv.org/abs/1711.02476.

[83] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Detectives: Detecting Coalition Hit Inflation Attacks in Advertising Networks Streams". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: Association for Computing Machinery, 2007, 241–250. ISBN: 9781595936547. DOI: 10.1145/1242572.1242606. URL: https://doi.org/10.1145/1242572.1242606.

[84] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. "Measurement and Analysis of Online Social Networks". In: *Proc. of the ACM Int. Conf. on Internet Measurement (SIGCOMM)*. 2007, 29–42.

[85] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store". In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, June 2013, pp. 103–114. ISBN: 978-1-931971-01-0. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell.

[86] Rasmus Pagh. "Locality-sensitive Hashing without False Negatives". In: *Proceedings of the 2016 Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1–9. DOI: 10.1137/1.9781611974331.ch1. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611974331.ch1. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611974331.ch1.

[87] Rasmus Pagh. “CoveringLSH: Locality-Sensitive Hashing without False Negatives”. In: *ACM Trans. Algorithms* 14.3 (June 2018). ISSN: 1549-6325. DOI: 10.1145/3155300. URL: https://doi.org/10.1145/3155300.

[88] Jean Paoli, François Yergeau, Michael Sperberg-McQueen, Tim Bray, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. https://www.w3.org/TR/2008/REC-xml-20081126/. W3C, Nov. 2008.

[89] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. Choudhary. “A new scalable parallel DBSCAN algorithm using the disjoint-set data structure”. In: *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11.

[90] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey. “Pardicle: Parallel Approximate Density-Based Clustering”. In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 560–571.

[91] Mateusz Pawlik and Nikolaus Augsten. “Tree edit distance: Robust and memory-efficient”. In: *Information Systems* 56 (2016), pp. 157 –173. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2015.08.004. URL: http://www.sciencedirect.com/science/article/pii/S0306437915001611.

[92] Mateusz Pawlik, Thomas Hütter, Daniel Kocher, Willi Mann, and Nikolaus Augsten. “A Link is not Enough - Reproducibility of Data”. In: *Datenbank-Spektrum* 19.2 (2019), pp. 107–115. DOI: 10.1007/s13222-019-00317-8. URL: https://doi.org/10.1007/s13222-019-00317-8.

[93] Ninh Pham and Rasmus Pagh. “Scalability and Total Recall with Fast CoveringLSH”. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM ’16. Indianapolis, Indiana, USA: Association for Computing Machinery, 2016, 1109–1118. ISBN: 9781450340731. DOI: 10.1145/2983323.2983742. URL: https://doi.org/10.1145/2983323.2983742.

[94] Martin Pichl, Eva Zangerle, and Günther Specht. “Combining Spotify and Twitter Data for Generating a Recent and Public Dataset for Music Recommendation”. In: *Proc. of the Workshop Grundlagen von Datenbanken*. Vol. 1313. CEUR Workshop Proceedings. 2014, pp. 35–40.

[95] Sven Puhlmann, Melanie Weis, and Felix Naumann. “XML Duplicate Detection Using Sorted Neighborhoods”. In: *International Conference on Extending Database Technology (EDBT)*. Vol. 3896. Lecture Notes in Computer Science. Springer, 2006, pp. 773–791. DOI: 10.1007/11687238_46.

[96] Jianbin Qin and Chuan Xiao. “Pigeonring: A Principle for Faster Thresholded Similarity Search”. In: *Proc. VLDB Endow.* 12.1 (Sept. 2018), 28–42. ISSN: 2150-8097. DOI: 10.14778/3275536.3275539. URL: https://doi.org/10.14778/3275536.3275539.

[97] Davi De Castro Reis, Paulo Braz Golgher, Altigran Soares Silva, and AlbertoF Laender. "Automatic web news extraction using tree edit distance". In: *Proceedings of the 13th international conference on World Wide Web*. 2004, pp. 502–511.

[98] Leonardo Andrade Ribeiro and Theo Härder. "Generalizing Prefix Filtering to Improve Set Similarity Joins". In: *Inf. Syst.* 36.1 (Mar. 2011), pp. 62–78. ISSN: 0306-4379. DOI: 10.1016/j.is.2010.07.003. URL: http://dx.doi.org/10.1016/j.is.2010.07.003.

[99] A. Sarma, P. Goyal, S. Kumari, A. Wani, J. S. Challa, S. Islam, and N. Goyal. "μDBSCAN: An Exact Scalable DBSCAN Algorithm for Big Data Exploiting Spatial Locality". In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–11.

[100] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN". In: *ACM Trans. Database Syst.* 42.3 (July 2017). ISSN: 0362-5915. DOI: 10.1145/3068335. URL: https://doi.org/10.1145/3068335.

[101] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 6th Edition. McGraw-Hill Higher Education, 2011. ISBN: 978-0-07-352332-3. URL: https://www.db-book.com/db6/index.html.

[102] *Simian - Similarity Analyzer | Duplicate Code Detection for the Enterprise*. https://www.harukizaemon.com/simian/. Accessed: 2019-02-11, 02:29 PM.

[103] R. Singhal. "Inside Intel® Core microarchitecture (Nehalem)". In: *2008 IEEE Hot Chips 20 Symposium (HCS)*. 2008, pp. 1–25. DOI: 10.1109/HOTCHIPS.2008.7476555.

[104] Hwanjun Song and Jae-Gil Lee. "RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 1173–1187. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3196887. URL: http://doi.acm.org/10.1145/3183713.3196887.

[105] Minseok Song, Christian W. Günther, and Wil M. P. van der Aalst. "Trace Clustering in Process Mining". In: *Business Process Management Workshops*. Ed. by Danilo Ardagna, Massimo Mecella, and Jian Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 109–120. ISBN: 978-3-642-00328-8.

[106] Volker Springel et al. "Simulations of the formation, evolution and clustering of galaxies and quasars". In: *Nature* 435.7042 (2005), pp. 629–636. ISSN: 1476-4687. DOI: 10.1038/nature03597. URL: https://doi.org/10.1038/nature03597.

[107] P. Stenström. "A Survey of Cache Coherence Schemes for Multiprocessors". In: *Computer* 23.6 (1990), pp. 12–24. DOI: 10.1109/2.55497.

[108] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. "Balance-aware Distributed String Similarity-based Query Processing System". In: *Proc. VLDB Endow.* 12.9 (May 2019), pp. 961–974. ISSN: 2150-8097. DOI: 10.14778/3329772.3329774. URL: https://doi.org/10.14778/3329772.3329774.

[109] Kuo-Chung Tai. "The Tree-to-Tree Correction Problem". In: *J. ACM* 26.3 (July 1979), 422–433. ISSN: 0004-5411. DOI: 10.1145/322139.322143. URL: https://doi.org/10.1145/322139.322143.

[110] MingJie Tang, Yongyang Yu, Walid G. Aref, Qutaibah M. Malluhi, and Mourad Ouzzani. "Efficient Processing of Hamming-Distance-Based Similarity-Search Queries Over MapReduce". In: *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.* Ed. by Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens. OpenProceedings.org, 2015, pp. 361–372. DOI: 10.5441/002/edbt.2015.32. URL: https://doi.org/10.5441/002/edbt.2015.32.

[111] Yu Tang, Yilun Cai, and Nikos Mamoulis. "Scaling Similarity Joins over Tree-structured Data". In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1130–1141. ISSN: 2150-8097. DOI: 10.14778/2809974.2809976. URL: http://dx.doi.org/10.14778/2809974.2809976.

[112] Robert Endre Tarjan. "Efficiency of a Good But Not Linear Set Union Algorithm". In: *J. ACM* 22.2 (Apr. 1975), 215–225. ISSN: 0004-5411. DOI: 10.1145/321879.321884. URL: https://doi.org/10.1145/321879.321884.

[113] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. "Efficient and Self-tuning Incremental Query Expansion for Top-k Query Processing". In: *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.* SIGIR '05. Salvador, Brazil: ACM, 2005, pp. 242–249. ISBN: 1-59593-034-5. DOI: 10.1145/1076034.1076077. URL: http://doi.acm.org/10.1145/1076034.1076077.

[114] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. "SpotSigs: Robust and Efficient near Duplicate Detection in Large Web Collections". In: *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.* SIGIR '08. Singapore, Singapore: Association for Computing Machinery, 2008, 563–570. ISBN: 9781605581644. DOI: 10.1145/1390334.1390431. URL: https://doi.org/10.1145/1390334.1390431.

[115] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. "Top-k Query Evaluation with Probabilistic Guarantees". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30.* VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 648–659. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316746.

[116] Martin Theobald, Holger Bast, Debapriyo Majumdar, Ralf Schenkel, and Gerhard Weikum. "TopX: efficient and versatile top-k query processing for semistructured data". In: *The VLDB Journal* 17.1 (2008), pp. 81–115. ISSN: 0949-877X. DOI: 10.1007/s00778-007-0072-z. URL: https://doi.org/10.1007/s00778-007-0072-z.

[117] Esko Ukkonen. "Approximate string-matching with q-grams and maximal matches". In: *Theoretical Computer Science* 92.1 (1992), pp. 191–211. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(92)90143-4. URL: https://www.sciencedirect.com/science/article/pii/0304397592901434.

[118] Rares Vernica, Michael J. Carey, and Chen Li. "Efficient Parallel Set-similarity Joins Using MapReduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 495–506. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807222. URL: http://doi.acm.org/10.1145/1807167.1807222.

[119] Márcio L. A. Vidal, Altigran S. da Silva, Edleno S. de Moura, and João M. B. Cavalcanti. "Structure-driven Crawler Generation by Example". In: *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '06. Seattle, Washington, USA: ACM, 2006, pp. 292–299. ISBN: 1-59593-369-7. DOI: 10.1145/1148170.1148223. URL: http://doi.acm.org/10.1145/1148170.1148223.

[120] P. Viswanath and V. Suresh Babu. "Rough-DBSCAN: A fast hybrid density based clustering method for large data sets". In: *Pattern Recognition Letters* 30.16 (2009), pp. 1477 –1488. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2009.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0167865509002153.

[121] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. 1st. USA: Apress, 2019. ISBN: 1484243978.

[122] Jiannan Wang, Guoliang Li, and Jianhua Feng. "Can We Beat the Prefix Filtering? An Adaptive Framework for Similarity Join and Search". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, 85–96. ISBN: 9781450312479. DOI: 10.1145/2213836.2213847. URL: https://doi.org/10.1145/2213836.2213847.

[123] Pei Wang, Chuan Xiao, Jianbin Qin, Wei Wang, Xiaoyang Zhang, and Yoshiharu Ishikawa. "Local Similarity Search for Unstructured Text". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, 1991–2005. ISBN: 9781450335317. DOI: 10.1145/2882903.2915211. URL: https://doi.org/10.1145/2882903.2915211.

[124] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. "Leveraging Set Relations in Exact Set Similarity Join". In: *Proc. VLDB Endow.* 10.9 (May 2017), pp. 925–936. ISSN: 2150-8097. DOI: 10.14778/3099622.3099624. URL: https://doi.org/10.14778/3099622.3099624.

[125] Y. Wang, D. J. DeWitt, and J. . Cai. "X-Diff: an effective change detection algorithm for XML documents". In: *Proceedings 19th International Conference on Data Engineering*. 2003, pp. 519–530. DOI: 10.1109/ICDE.2003.1260818.

[126] Yiqiu Wang, Yan Gu, and Julian Shun. "Theoretically-Efficient and Practical Parallel DBSCAN". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 2555–2571. ISBN: 9781450367356. DOI: 10.1145/3318464.3380582. URL: https://doi.org/10.1145/3318464.3380582.

[127] Anthony Williams. *C++ Concurrency in Action*. Vol. 2. Manning Publications, 2019.

[128] Y. Wu, J. Guo, and X. Zhang. "A Linear DBSCAN Algorithm Based on LSH". In: *2007 International Conference on Machine Learning and Cybernetics*. Vol. 5. 2007, pp. 2608–2614.

[129] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. "Top-k Set Similarity Joins". In: *Proceedings of the 2009 IEEE International Conference on Data Engineering*. ICDE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 916–927. ISBN: 978-0-7695-3545-6. DOI: 10.1109/ICDE.2009.111. URL: http://dx.doi.org/10.1109/ICDE.2009.111.

[130] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. "Efficient Similarity Joins for Near-Duplicate Detection". In: *ACM Trans. Database Syst.* 36.3 (Aug. 2011). ISSN: 0362-5915. DOI: 10.1145/2000824.2000825. URL: https://doi.org/10.1145/2000824.2000825.

[131] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. "A Fast Parallel Clustering Algorithm for Large Spatial Databases". In: *Data Min. Knowl. Discov.* 3.3 (Sept. 1999), 263–290. ISSN: 1384-5810. DOI: 10.1023/A:1009884809343. URL: https://doi.org/10.1023/A:1009884809343.

[132] K. Yang, Y. Gao, R. Ma, L. Chen, S. Wu, and G. Chen. "DBSCAN-MS: Distributed Density-Based Clustering in Metric Spaces". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 1346–1357.

[133] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. "Similarity Evaluation on Tree-structured Data". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 754–765. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066243. URL: http://doi.acm.org/10.1145/1066157.1066243.

[134] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. "The End of a Myth: Distributed Transactions Can Scale". In: *Proc. VLDB Endow.* 10.6 (Feb. 2017), pp. 685–696. ISSN: 2150-8097. DOI: 10.14778/3055330.3055335. URL: https://doi.org/10.14778/3055330.3055335.

[135] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. "Rethinking Database High Availability with RDMA Networks". In: *Proc. VLDB Endow.* 12.11 (July 2019), 1637–1650. ISSN: 2150-8097. DOI: 10.14778/3342263.3342639. URL: https://doi.org/10.14778/3342263.3342639.

[136] K. Zhang and D. Shasha. "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems". In: *SIAM J. Comput.* 18.6 (Dec. 1989), pp. 1245–1262. ISSN: 0097-5397. DOI: 10.1137/0218082. URL: http://dx.doi.org/10.1137/0218082.

[137] Kaizhong Zhang, Rick Statman, and Dennis Shasha. "On the editing distance between unordered labeled trees". In: *Information Processing Letters* 42.3 (1992), pp. 133–139. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(92)90136-J. URL: https://www.sciencedirect.com/science/article/pii/002001909290136J.

[138] Zijian Zheng, Ron Kohavi, and Llew Mason. "Real World Performance of Association Rule Algorithms". In: *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD).* 2001, 401–406.

[139] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. "JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes". In: *Proceedings of the 2019 International Conference on Management of Data.* SIGMOD '19. Amsterdam, Netherlands: ACM, 2019, pp. 847–864. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3300065. URL: http://doi.acm.org/10.1145/3299869.3300065.

[140] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. "Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks". In: *Proceedings of the 2019 International Conference on Management of Data.* SIGMOD '19. Amsterdam, Netherlands: ACM, 2019, pp. 741–758. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3300081. URL: http://doi.acm.org/10.1145/3299869.3300081.